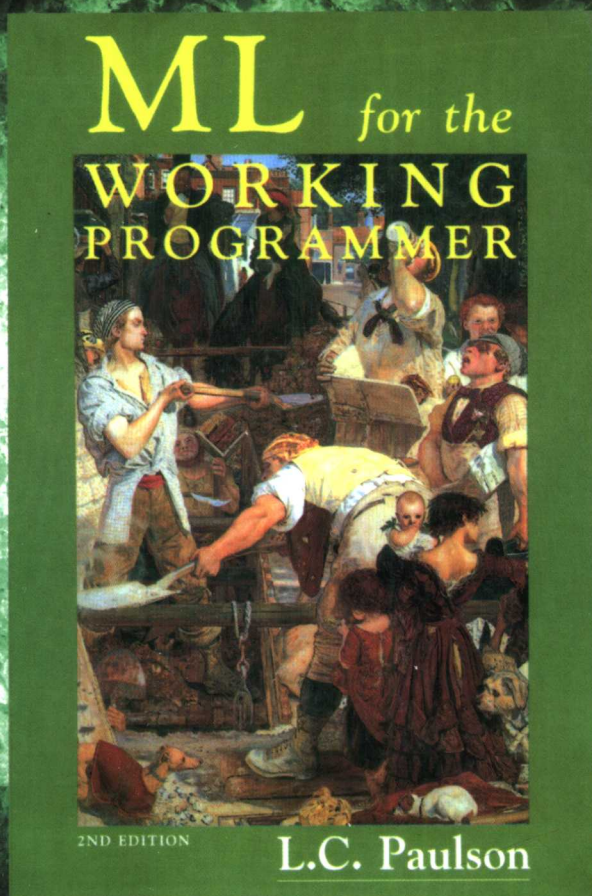


计 算 机 科 学 丛 书

原书第2版

ML程序设计教程

(英) Lawrence C. Paulson 著 柯 韦 译



ML for the Working Programmer
Second Edition



机械工业出版社
China Machine Press

本书是关于ML程序设计的经典教材，详细介绍如何使用ML语言进行程序设计，并讲解函数式程序设计的基本原理。

书中含有大量例子，涵盖了排序、矩阵运算、多项式运算等方面。大型的例子包括一个一般性的自顶向下语法分析器、一个 λ -演算归约程序和一个定理证明机。书中也讲述了关于数组、队列、优先队列等高效的函数式实现，并且有一章专门讨论函数式程序的形式论证。本书的代码均可以从作者网站 (<http://www.cl.cam.ac.uk/users/lcp/>) 得到。

作者简介

Lawrence C. Paulson

于1981年在美国斯坦福大学获得计算机科学博士学位，现为英国剑桥大学计算逻辑学教授。Paulson博士从事有关ML语言的教学和工作多年，拥有扎实的背景和丰富的经验，并曾经参与Standard ML的设计。Paulson博士开发和维护了Isabelle自动定理证明系统，他近期正在进行关于自动定理证明和密码协议验证方面的研究。



ISBN 7-111-16121-1



华章图书

华章网站 <http://www.hzbook.com>

网上购书: www.china-pub.com

北京市西城区百万庄南街1号 100037

读者服务热线: (010)68995259, 68995264

读者服务信箱: hzedu@hzbook.com

ISBN 7-111-16121-1/TP · 4199

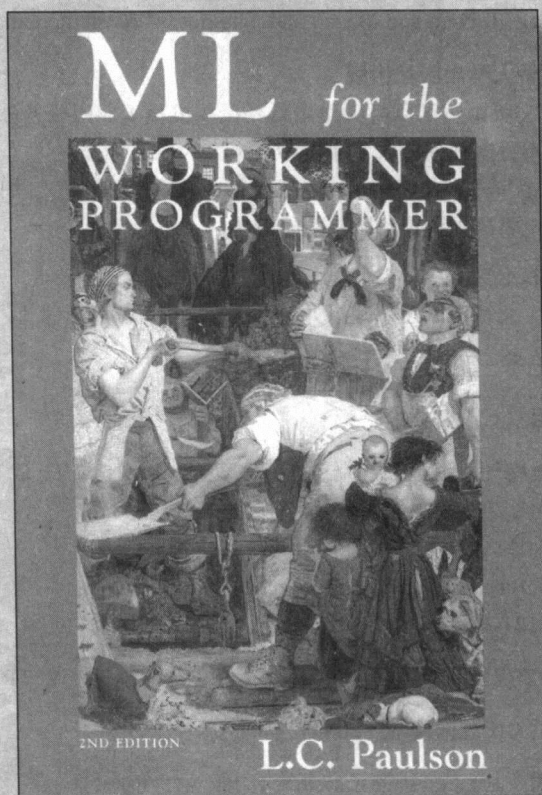
定价: 45.00 元

计 算 机 科 学 丛 书

原书第2版

ML程序设计教程

(英) Lawrence C. Paulson 著 柯 韦 译



ML for the Working Programmer
Second Edition



机械工业出版社
China Machine Press

本书详细讲解如何使用ML语言进行程序设计，并介绍函数式程序设计的基本原理。书中特别讲述了为ML的修订版所设计的新标准库的主要特性，并且给出大量例子，涵盖排序、矩阵运算、多项式运算等方面。大型的例子包括一个一般性的自顶向下语法分析器、一个 λ -演算归约程序和一个定理证明机。书中也讲述了关于数组、队列、优先队列等高效的函数式实现，并且有一章专门讨论函数式程序的形式论证。

本书可作为高等院校计算机专业相关课程的教材，也适合广大程序设计人员参考。

Lawrence C. Paulson: ML for the Working Programmer, 2nd edition.

Originally published by Cambridge University Press in 2001.

This Chinese edition is published with the permission of the Syndicate of the Press of the University of Cambridge, Cambridge, England.

Copyright © 2001 by Cambridge University Press.

This edition is licensed for distribution and sale in the People's Republic of China only, excluding Hong Kong, Taiwan and Macao and may not be distributed and sold elsewhere.

本书原版由剑桥大学出版社出版。

本书简体字中文版由英国剑桥大学出版社授权机械工业出版社独家出版。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

此版本仅限在中华人民共和国境内（不包括中国香港、台湾、澳门地区）销售发行，未经授权的本书出口将被视为违反版权法的行为。

版权所有，侵权必究。

本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2004-3938

图书在版编目（CIP）数据

ML程序设计教程（原书第2版）/（英）保罗森（Paulson, L. C.）著；柯韦译. —北京：机械工业出版社，2005.5

（计算机科学丛书）

书名原文：ML for the Working Programmer, 2nd edition

ISBN 7-111-16121-1

I. M … II. ①保…②柯… III. 程序语言—程序设计 IV. TP312

中国版本图书馆CIP数据核字（2005）第017060号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：朱起飞 冯春丽

北京瑞德印刷有限公司印刷·新华书店北京发行所发行

2005年5月第1版第1次印刷

787mm × 1092mm 1/16 · 24.25印张

印数：0 001-3000册

定价：45.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：（010）68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅肇划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall, Addison-Wesley, McGraw-Hill, Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum, Stroustrup, Kernighan, Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及度藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域

的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业的教学度身订造的。其中许多教材均已为M. I. T., Stanford, U.C. Berkeley, C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方式如下：

电子邮件: hzedu@hzbook.com

联系电话: (010) 68995264

联系地址: 北京市西城区百万庄南街1号

邮政编码: 100037

专家指导委员会

(按姓氏笔画顺序)

尤晋元
石教英
张立昂
邵维忠
周立柱
范明
袁崇义
谢希仁

王珊
吕建
李伟琴
陆丽娜
周克定
郑国梁
高传善
裘宗燕

冯博琴
孙玉芳
李师贤
陆鑫达
周傲英
施伯乐
梅宏
戴葵

史忠植
吴世忠
李建中
陈向群
孟小峰
钟玉琢
程旭

史美林
吴时霖
杨冬青
周伯生
岳丽华
唐世渭
程时端

译者序

多年前我曾经认为自己是一个程序员，对程序设计有着执著的兴趣。试图洞悉本质的渴望以及相关知识的匮乏往往令我陷入苦思。于是 I 开始了新一轮的阅读。在一个图书馆中书架林立的某处，我发现了这本写给程序员的书。在我相对贫乏的阅读历史中，书名带有“程序员”（programmer）的寥寥无几，又怎能不留意呢？这本书最终没有让我失望。

· 如果说过去我对程序设计有过一些有意义的思考，而从中得到了乐趣的话，函数式程序设计则系统地叙述了这些令人兴奋的闪光之处。这种基于某种数学概念的方法把我们从程序运行的细节中解放出来，转而关注如何去表达。程序因此变成一个个静态的方程式，安详地躺在那里，展示着简单而丰富的内涵。我们则不再被传统的动态步骤困扰，思想在清晰的、静态的表达基础上轻松地延伸着，这是多么美妙的感受啊！我们曾经认为，如果无法理解赋值语句就不能进行程序设计，我们费了多少力气去习惯它，以至于后来已完全在脑海中根深蒂固了。现在我们要从这种被扭曲的思维中走出来，回到更为自然的思考方式去，而函数式程序设计则是众多方法中的一个。

ML语言是函数式语言中较为经典的，它的语法简单优美，语义清晰准确并易于理解，语言的实现也相当高效和严谨。和最新式的函数式语言不同，ML牺牲了一些数学上的纯粹性，换来了相对简单的语义描述，理解它不需要十分高深的数学抽象思考。这对于语言的实用性是十分重要的。正如书中所说，我们不希望从机器语言繁琐的运行步骤中走出来，而立即又掉进同样复杂繁琐的数学模型中去。作为一种入门的函数式程序设计语言，ML不仅对于初学程序设计的学生是一个好的语言，对于有经验的程序员来说也是一个好的桥梁，程序员们将或多或少地发现，他们过去的一些思考竟有如此简单的归宿。我想，这也是原书命名的由来。

原书是一本非常流行的ML语言和函数式程序设计教材，尽管第2版面世至今已有很长时间了，但依旧被国外许多大学的相关科目广泛采用。书中在介绍ML语言的同时更多地阐述了语言和函数式程序设计的思考方法。大量的例子条理分明，深浅搭配适度。随处可见的精心安排的旁白涉及了计算机语言理论研究的许多方面，作为科普读物来说也非常赏心悦目。我很喜欢这本书，衷心地希望能将之介绍给更多的中国读者。我的翻译工作是认真而仔细的，然而由于水平有限，令人遗憾之处恐在所难免，希望读者不要因此受阻，能在阅读中有所收获。

我感谢机械工业出版社，她们出版了许多计算机基础理论的译著，这一本也不例外。我衷心希望此举能将计算机不仅作为一种技术，也作为一种科学推广，产生更多的思想积淀，形成更广泛的基础。我还要感谢在此书翻译过程中给我宝贵指点和帮助的各位尊敬的老师和编辑们。原书的作者Paulson博士也对我的提问作出了耐心的讲解。

第2版序

每次重新印刷，书中的一些小错误都会悄悄地消失。但是重新印刷并不会做大的改进，尽管改进是很有意义的。因为这样做会影响页码编号，使内容产生差别而互不兼容。重大的改动积累起来（以及编辑的催促）便产生了这个第2版。

非常幸运的是，对ML语言的改动都凑到了一起。ML有了新的标准库，并且语言本身也经过了修订。值得强调的是，这些改动没有牺牲ML固有的可靠性。一些晦涩的技术要点已经得到简化，原先定义中不合适的地方也改正了。现有的程序几乎不用改动就可以继续运行。最明显的改动是增加了新的字符类型，还有一套新的顶层库函数。

这版新书及时反映了语言的变化，并在格局安排上做了很大的改进。模块提前到第2章就介绍了，而不是放到第7章，它的使用贯穿全书。这使得重点有所转移，从数据结构（比如二叉搜索树）变为抽象类型（比如字典）。抽象类型通常在某一小节引入，并给出它的ML签名。然后讲解实现背后的思想，最后给出ML结构的代码。虽然评审对第1版较为宽容，但是许多读者要求重新组织内容。

书中的程序不仅移动了位置，而且重新编写过。它们反映了如何使用模块的新思路。由于`open`声明会隐藏模块结构，所以已经很少出现了。函子也仅在需要的时候才使用。现在，程序的缩进也排得很仔细了，加上其他改进，代码变得更加易读，质量也更好了，表现在合并排序更为简单迅速，优先队列也更快了。

新标准库也要求尽早地提到模块。尽管这样做就必须修改现有代码，但它能使ML在现实语言中的地位更加稳固。新标准库的设计经过了漫长的磋商过程，主要是为了提供全面的支持，同时避免过分的复杂化。它的组织显示了ML模块的优点。字符串处理、输入输出和系统接口这些模块都提供了实际的功能改进。

新标准库导致很多代码必须重写。当新标准库包含函数`foldl`时，读者一般不愿意再看到以前类似的函数`foldleft`。但是这些函数不是完全一样的，所以，重写并不只是改个名字那么简单。很多曾讲述实用函数的章节，现在都要对照新标准结构进行检查更新。

更新过的参考文献说明了函数式程序设计和ML在各领域的广泛应用。ML符合构建可靠系统的要求。软件工程师们需要一种能提供类型安全、模块化、编译时一致性检测和容错（异常）的语言。ML程序是可移植的，这要部分归功于标准库。商业化的编译器正在不断提高质量和效率。ML的运行速度可以和C媲美，特别是在需要复杂存储管理的应用场合。本书的名字（指英文书名）曾引来一些嘲笑，但却给出了很好的提示。

我最惊讶的是看到第1版出现在初级程序员的手中，尽管第一页上写着让他们看点别的书。为了帮助初学者，我加入了一些特别简单的例子，并将大多数参考引用从正文中移走。重写了第1章，试图以一种既适合初学者又适合有经验的C程序员的方式介绍基本的程序设计概念。这比听上去要容易：C并不想给程序员一个解决问题的环境，而仅仅是给底层的硬件稍加装扮。第1章仍旧包含一些基本的计算机常识，但教师也许还是喜欢从第2章开始讲述，因为它带有

简单的会话过程。

在书的末尾列出了一些项目建议。我故意说得不很明确，因为一个大项目的第一步就是准确地分析需求。我希望看到越来越多的人在项目中采用ML，选择ML，特别是取代像C这样不安全的语言，最终会被看作是专业的一种标志。

我非常感谢所有对这一版给出有用的意见、建议或代码的人们。他们分别是Matthew Arcus、Jon Fairbairn、Andy Gordon、Carl Gunter、Michael Hansen、Andrew Kennedy、David MacQueen、Brian Monahan、Arthur Norman、Chris Okasaki、John Reppy、Hans Rischel、Peter Sestoft、Mark Staples和Mads Tofte。Sestoft还给了我一个Moscow ML的预发行版，含有库的更新。CUP的Alison Woollatt编写了 \LaTeX 的类文件。Franklin Chen和Namhyun Hur报告了前一版中的错误。

前言

本书源于对Standard ML和函数式程序设计的讲稿。它仍可以作为函数式程序设计的课本——一本面向实用，而不是标准的、理想化的书——然而，它主要是一本有效使用ML的指南。它甚至讨论了ML的命令式特性。

有些内容需要离散数学的知识，例如初等逻辑和集合论。读者会发现以往的程序设计经验是有用的，但不是必需的。

本书是一本程序设计手册，而不是参考手册。它覆盖了ML的主要方面，但并不尽述所有的细节。它在理论原理上花费了一些篇幅，但主要还是关心高效的算法和实际的程序设计。

本书的组织反映了我的教学经验。高阶函数出现得较晚，在第5章讲述。惯常的做法是在一开始就介绍一些不甚自然的例子，这样做只能使学生们感到困惑。高阶函数的概念是不容易理解的，需要充分的预备知识。所以，本书从基本类型、表和树开始讲述。当讲到高阶函数时，很多相关的例子已经是现成的了。

练习的难度相差很大。它们不是用来评测学生的，而是为了提供实践机会，拓展内容和激发讨论的。


本书一览。大多数章节都专注于ML的各个方面。第1章介绍了函数式程序设计的背景思想，以及ML的历史概况。第2~5章涵盖了ML的函数式部分，包括对模块的简介。讲述了基本类型、表、树和高阶函数。对函数式程序设计的更广泛的原理也有所讨论。

第6章给出了论证函数式程序的形式方法。看上去似乎偏离了程序设计的主题，然而错误的程序是没用的。易于形式论证是函数式程序设计的一大好处。

第7章详细讲述了模块，包括函子（带参数的模块）。第8章讲述了ML的命令式特性：引用、数组和输入输出。本书的其余部分由较大的例子构成。第9章给出了函数式的语法分析器和一个 λ -演算解释器。第10章给出了一个定理证明机，这是ML的传统应用。

书中的例子非常丰富。其中一些只是为了说明ML的某个方面，但大多数本身就有一定用途——排序、函数式数组、优先队列、搜索算法、美化打印。请注意：虽然我测试过这些程序，但是它们仍不免含有错误。


信息和警告块。技术性的旁白、库函数的叙述以及为进一步学习而给出的笔记都会不时地出现。它们被加以如下图标以便有些读者可以跳过：

 亨利王的要求。他们拿不出什么理由可以反对陛下向法兰西提出王位的要求，只除了这一点，那个在法拉蒙时代制定的一条法律，*In terram Salicam mulieres ne succedant*，‘在撒利族的土地上妇女没有继承权’：而法国人就把这‘撒利族的土地’曲解为法兰西的土地，并且把法拉蒙认做是这条法律的创制人和妇权的剥夺者。可是他们的历史学家却忠实地宣称撒利区是在日耳曼的土地上……[⊖]

⊖ 书中的技术性旁白不会像这位大主教的演讲那么长，它有62行。

（两段译文分别出自莎士比亚《亨利五世》和《理查三世》中译本。——译者注）

ML并不完美。某些缺陷会使简单的编码错误浪费掉程序员几个小时的时间。而且，新的标准库使得新旧编译器不兼容。因此，本书中有一些这样的警告图标：

 小心葛罗斯特公爵。呵，勃金汉！小心那个狗东西：要知道，摇尾的狗会咬人；咬了人，它的牙毒还会叫你痛极而死；莫同他来往，千万留意；罪恶、死亡和地狱都看中了他，地下的大小役吏都在供他使唤。

我要赶紧补充一点，在ML里不会产生这么可怕的后果。程序里的错误是不能冲垮ML系统本身的。另一方面，程序员必须牢记，即使是正确的程序也可能给外部世界带来伤害。

如何得到Standard ML编译器。由于Standard ML刚出现不久，很多学院没有编译器。下面列出了现有的一些Standard ML编译器，并附有联系地址。书中的例子是在Moscow ML、Poly/ML和Standard ML of New Jersey下开发的。我尚未尝试其他的编译器。

要得到MLWorks，请联系Harlequin Limited, Barrington Hall, Barrington, Cambridge, CB2 5RG, England。他们的电子邮件地址是web@harlequin.com。

要得到Moscow ML，请联系Peter Sestoft, Mathematical Section, Royal Veterinary and Agricultural University, Thorvaldsensvej 40, DK-1871 Frederiksberg C, Denmark。或从互联网上得到该系统：

<http://www.dina.kvl.dk/~sestoft/mosml.html>

要得到Poly/ML，请联系Abstract Hardware Ltd, 1 Brunel Science Park, Kingston Lane, Uxbridge, Middlesex, UB8 3PQ, England。他们的电子邮件地址是lambda@ahl.co.uk。或从互联网上得到该系统[⊖]：

<http://www.polymml.org/>

要得到Poplog Standard ML，请联系Integral Solutions Ltd, Berk House, Basing View, Basingstoke, Hampshire RG21 4RG, England。他们的电子邮件地址是isl@isl.co.uk。

要得到Standard ML of New Jersey，请联系Andrew Appel, Computer Science Department, Princeton University, Princeton NJ 08544-2087, USA。更好的是可以从互联网上得到文件[⊖]：

<http://www.cs.princeton.edu/~appel/smlnj/>

<http://www.smlnj.org/>

书中的程序和一些练习答案可以通过电子邮件得到，我的电子邮件地址是lcp@cl.cam.ac.uk。如果可能，请使用互联网，我的主页在

<http://www.cl.cam.ac.uk/users/lcp/>

致谢。编辑，David Tranah，在写作的各个阶段提供了帮助，并建议了书名。Graham Birtwistle、Glenn Bruns和David Wolfram仔细阅读了文本。Dave Berry、Simon Finn、Mike Fourman、Kent Karlsson、Robin Milner、Richard O'Keefe、Keith van Rijsbergen、Nick Rothwell、Mads Tofte、David N. Turner和Harlequin的工作人员也对文本提出了意见。Andrew Appel、Gavin Bierman、Phil Brabbin、Richard Brooksby、Guy Cousineau、Lal George、Mike Gordon、Martin Hansen、Darrell Kindred、Silvio Meira、Andrew Morris、Khalid Mughal、

⊖ ⊖ 这两个网址原书没有，中译本加上的。——译者注

Tobias Nipkow、Kurt Olender、Allen Stoughton、Reuben Thomas、Ray Toal和Helen Wilson发现了前几次印刷中的错误。Piete Brooks、John Carroll和Graham Titmus在计算机使用方面给予了帮助。我还要感谢Dave Matthews开发了Poly/ML，这是多年以来唯一高效的Standard ML的编译器。

在众多的参考文献中，Abelson和Sussman（1985）、Bird和Wadler（1988）以及Burge（1975）的著作特别有帮助。Reade（1989）的书中包含了在ML中实现惰性表的有用思想。

The Science and Engineering Research Council在过去20多年来给予了LCF和ML大量的研究资助。

本书的大部分写作工作都是我从剑桥大学休假的过程中完成的。我感谢计算机实验室（Computer Laboratory）和卡莱尔学院（Clare College）给予休假，以及爱丁堡大学对我六个月的招待。

最后，我要感谢Sue，感谢她所给我的一切帮助，以及天天耐心倾听我关于每一章进展的报道。

目 录

出版者的话	
专家指导委员会	
译者序	
第2版序	
前言	
第1章 Standard ML	1
函数式程序设计	2
1.1 表达式和命令	2
1.2 过程式程序设计语言中的表达式	3
1.3 存储管理	3
1.4 函数式语言的元素	4
1.5 函数式程序设计的效率	7
Standard ML概述	8
1.6 Standard ML的演化	8
1.7 ML的自动定理证明传统	9
1.8 新标准库	10
1.9 ML和工作中的程序员	11
第2章 名字、函数和类型	13
本章提要	13
值的声明	14
2.1 命名常量	14
2.2 声明函数	15
2.3 Standard ML中的标识符	16
数、字符串和真值	17
2.4 算术运算	17
2.5 字符串和字符	19
2.6 真值和条件表达式	20
序偶、元组和记录	21
2.7 向量: 序偶的例子	21
2.8 多参数和多结果的函数	22
2.9 记录	24
2.10 中缀操作符	27
表达式的求值	29
2.11 ML中的求值: 传值调用	29
2.12 传值调用下的递归函数	30
2.13 传需调用或惰性求值	33
书写递归函数	36
2.14 整数次幂	36
2.15 斐波那契数列	37
2.16 整数平方根	39
局部声明	39
2.17 例子: 实数平方根	40
2.18 使用local来隐藏声明	41
2.19 联立声明	42
模块系统初步	44
2.20 复数	44
2.21 结构	45
2.22 签名	46
多态类型检测	47
2.23 类型推导	47
2.24 多态函数声明	48
要点小结	50
第3章 表	51
本章提要	51
表的简介	51
3.1 表的构造	52
3.2 表的操作	53
基本的表函数	54
3.3 表的测试和分解	55
3.4 与数量有关的表处理	56
3.5 追加和翻转	58
3.6 表的表, 序偶的表	60
表的应用	61
3.7 找零钱	61
3.8 二进制算术	63
3.9 矩阵的转置	64

3.10 矩阵乘法	66	4.14 字典	113
3.11 高斯消元法	67	4.15 函数式数组和弹性数组	116
3.12 分解一个数为两个平方数之和	70	4.16 优先队列	120
3.13 求后继排列的问题	71	重言式检测器	124
多态函数中的相等测试	72	4.17 命题逻辑	124
3.14 相等类型	73	4.18 否定范式	126
3.15 多态集合操作	73	4.19 合取范式	127
3.16 关联表	76	要点小结	129
3.17 图的算法	77	第5章 函数和无穷数据	131
排序: 案例研究	81	本章提要	131
3.18 随机数	81	作为值的函数	131
3.19 插入排序	82	5.1 使用fn记法的匿名函数	132
3.20 快速排序	83	5.2 柯里函数	132
3.21 合并排序	84	5.3 数据结构中的函数	135
多项式算术	86	5.4 作为参数和结果的函数	135
3.22 表示抽象数据	87	通用算子	137
3.23 多项式的表示	87	5.5 切片	137
3.24 多项式加法和乘法	88	5.6 组合子	138
3.25 最大公因式	90	5.7 表算子map(映射)和filter(过滤)	139
要点小结	91	5.8 表算子takewhile和dropwhile	141
第4章 树和具体数据	93	5.9 表算子exists(存在)和all(全称)	141
本章提要	93	5.10 表算子foldl(左折叠)和foldr (右折叠)	142
数据类型声明	93	5.11 更多递归算子的例子	144
4.1 国王和他的臣民	94	序列, 或无穷表	147
4.2 枚举类型	95	5.12 序列类型	147
4.3 多态数据类型	97	5.13 基本的序列处理	149
4.4 通过val、as、case进行模式匹配	99	5.14 基本的序列应用	151
异常	101	5.15 数值计算	153
4.5 异常初步	101	5.16 交替和序列的序列	155
4.6 声明异常	102	搜索策略和无穷表	156
4.7 抛出异常	103	5.17 用ML实现的搜索策略	158
4.8 处理异常	105	5.18 生成回文	159
4.9 对异常的异议	106	5.19 八皇后问题	160
树	107	5.20 迭代深化	161
4.10 二叉树类型	107	要点小结	162
4.11 枚举树的内容	109	第6章 函数式程序的论证	163
4.12 由表建立树	111	本章提要	163
4.13 为二叉树设计的结构	112	一些数学证明的原理	163
基于树的数据结构	112		

6.1 ML程序和数学	164	7.17 模块声明的语法	237
6.2 数学归纳法和完全归纳法	165	要点小结	237
6.3 程序验证的简单例子	168	第8章 ML中的命令式程序设计	239
结构归纳法	171	本章提要	239
6.4 关于表的结构归纳法	171	引用类型	239
6.5 关于树的结构归纳法	175	8.1 引用及其操作	240
6.6 函数值和算子	178	8.2 控制结构	242
一般性归纳原理	181	8.3 多态引用	245
6.7 计算范式	182	数据结构中的引用	249
6.8 良基归纳和递归	185	8.4 序列, 或惰性表	249
6.9 递归程序模式	187	8.5 环形缓冲区	252
描述和验证	189	8.6 可变更的数组和函数式的数组	255
6.10 有序谓词	190	输入和输出	259
6.11 通过多重集合表示重新排列	191	8.7 字符串处理	259
6.12 验证的意义	194	8.8 文本输入输出	262
要点小结	195	8.9 文本处理的例子	264
第7章 抽象类型和函子	197	8.10 美化打印程序	267
本章提要	197	要点小结	271
队列的三种表示方法	198	第9章 书写 λ -演算的解释器	273
7.1 将队列表示为表	198	本章提要	273
7.2 将队列表示为新的数据类型	199	函数式语法分析器	273
7.3 将队列表示为表的序偶	200	9.1 扫描或词法分析	273
签名和抽象	201	9.2 自顶向下的语法分析套件	275
7.4 队列应具有签名	202	9.3 语法分析器的ML代码	277
7.5 签名约束	202	9.4 例子: 分析和显示类型	280
7.6 抽象类型 (abstype) 声明	204	λ -演算简介	284
7.7 从结构导出的签名	206	9.5 λ -项和 λ -归约	284
函子	207	9.6 在替换中防止变量的捕获	286
7.8 测试多个队列结构	208	在ML中表示 λ -项	288
7.9 泛型矩阵运算	210	9.7 基本操作	288
7.10 泛型的字典和优先队列	214	9.8 λ -项的语法分析	290
利用模块建立大型系统	217	9.9 显示 λ -项	291
7.11 多参数函子	217	作为程序设计语言的 λ -演算	293
7.12 共享约束	221	9.10 λ -演算中的数据结构	293
7.13 全函子式程序设计	224	9.11 λ -演算中的递归定义	296
7.14 open声明	228	9.12 λ -项的求值	296
7.15 签名和子结构	232	9.13 演示求值程序	299
模块参考指南	234	要点小结	301
7.16 签名和结构的语法	235		

第10章 策略定理证明机	303	10.11 命题策略	323
本章提要	303	10.12 量词策略	324
一阶逻辑的相继式演算	303	搜索证明	326
10.1 命题逻辑的相继式演算	304	10.13 变换证明状态的命令	326
10.2 证明相继式演算中的定理	305	10.14 两个使用策略的证明实例	328
10.3 量词的相继式规则	307	10.15 策略算子	330
10.4 带量词的定理证明	308	10.16 一阶逻辑的自动策略	333
在ML中处理项和公式	310	要点小结	336
10.5 表示项和公式	310	项目建议	337
10.6 分析和显示公式	312	参考文献	339
10.7 合一	316	Standard ML语法图	347
策略和证明状态	319	语法图中英词汇对照表	357
10.8 证明状态	319	索引	359
10.9 ML签名	320	预定义标识符	367
10.10 用于基本相继式的策略	321		

第1章 Standard ML

第一个ML编译器是在1974年实现的。随着用户群的成长，各种语言的变体开始出现。于是，ML社群便集中起来开发和普及一种通用的语言，Standard ML，有时简称为SML，或干脆就叫ML。现在可以找到一些很好的Standard ML编译器。

没用多久Standard ML就变得相当流行。世界各地都有大学采用它作为教授学生的第一门程序设计语言。开发者选择它作为一些具有相当规模的项目的实现语言。也许对于这种流行的初步解释是：用ML很容易写出清晰、可靠的程序。要得到更满意的答案，首先要分析一下我们是怎样看待计算机系统的。

计算机是非常复杂的。在一部典型的工作站里面所包括的硬件和软件就远不是一个人所能完全掌握的。不同的人对工作站有不同层次的理解。对于一般的用户，工作站是一部文字处理机或电子表格。对于维修工来说，工作站是一个盒子，里面有电源和电路板等。对于机器语言程序员，工作站则提供了一个巨大的字节存储器，连接在一个能够进行算术和逻辑运算的处理器上面。而应用程序员则会借助他所选择的程序设计语言作为媒介来理解工作站。

这里，我们把“电子表格”、“电源”和“处理器”都理解为理想的、抽象的概念。我们只考虑它们的功能和局限，却不关心它们是如何构造的。通过良好的抽象，我们可以有效地使用计算机，而不会被它的复杂性拖垮。

普通的高级程序设计语言在抽象的层次上并没有比机器语言高多少。这些语言提供了方便的记号，但仅限于那些可以被直接映射到机器码上去的操作。程序中一个小的错误可以破坏其他数据，甚至是程序本身。这种行为的结果若能解释的话也只能是在机器语言的层次上进行。

ML则高出机器语言层很多。它支持函数式程序设计 (functional programming)，其中，程序是由函数所组成的，这些函数操作简单的数据结构。函数式程序设计在许多方面对于解决问题都是非常理想的，下面会对这个问题进行简单的讨论，也会贯穿全书来进行演示。程序设计任务可以通过数学方式来达成，而不用事先知道计算机的内部工作情况。ML也提供可变的 (mutable) 变量和数组。可变的对象可以通过赋值语句来改变，利用这些，可以很容易表达任何传统的代码。为了构造大的系统，ML提供了模块 (module)，程序的某一部分可以分别描述和编码。

最为重要的是，ML可以防止程序员犯错误。在程序可以运行之前，编译器会检测所有模块的接口是否彼此相容，以及所有的数据是否被一致地使用。例如，一个整数不会被用作存储地址。(一个真正的程序不应该依赖于这种技巧。) 在程序运行中，更进一步的检测保证了安全性：甚至是一个错误的ML程序也会表现得像一个ML程序。它可能会永远运行下去，或者返回一个错误信息给用户。但是它不会崩溃。

ML支持一种面向程序员要求的抽象层次，而不是面向硬件的。ML系统可以保证这种抽象，即使程序是错误的。其他的程序设计语言很难提供这种保障。

函数式程序设计

程序设计语言有很多种风格。像Fortran、Pascal和C这样的语言被称为是过程式 (procedural) 的：它们主要的程序设计单元是过程。目前流行的优化程序设计方法主要是面向对象的，这些对象包含着与它们自身相关的那些操作。这种面向对象 (object-oriented) 的语言包括C++和Modula-3。这两种实现方法都是基于命令的，这些命令操作于机器状态上，它们都是命令式 (imperative) 的方法。

如同过程式语言是面向命令的那样，函数式语言则是面向表达式的。没有命令的程序设计对一些读者来说可能很奇怪，因此让我们看看这个想法的背后到底是什么，就从对命令式程序设计的批评开始。

1.1 表达式和命令

第一个高级程序设计语言Fortran给程序员们提供了算术表达式。他们不再需要对寄存器的加法和存取操作序列进行编码了，公式翻译器 (FORMula TRANslator) 已经替他们做了。为什么表达式如此重要？不是因为它们广为人知：其实对于下面公式

2

$$\sqrt{\frac{\sin^2 \theta}{1 + |\cos \phi|}}$$

Fortran的语法和它仅是略有相似而已。让我们还是详细看一下表达式的优点。在Fortran中表达式可以有副作用 (side effect)：它们可以改变状态。而下面我们重点讲解的是仅计算一个值的纯表达式。

表达式具有一种递归的结构。像下面这样的典型表达式

$$f(E_1 + E_2) - g(E_3)$$

是由其他表达式 E_1 、 E_2 和 E_3 组成的，它自己也可以成为更大的表达式的一部分。表达式的值是递归地由它的子表达式给出的。这些子表达式可以以任意的顺序，甚至是并行地进行求值。

表达式可以利用数学定律来进行变形。例如，把 $E_1 + E_2$ 替换成 $E_2 + E_1$ 并不会影响上面表达式的值，这要归功于加法的交换律。这种利用相等的项去进行替换的能力称为引用透明 (referential transparency)，特别是一个表达式可以被安全地替换成它的值。

命令也具有很多类似的优点。在现代语言中，命令是由其他命令构成的。像下面的一个命令

```
while  $B_1$  do (if  $B_2$  then  $C_1$  else  $C_2$ )
```

的含义可以由它的各个组成部分的含义给出。命令也可以享受引用透明：像定律

```
(if  $B$  then  $C_1$  else  $C_2$ );  $C \equiv$  if  $B$  then ( $C_1$ ;  $C$ ) else ( $C_2$ ;  $C$ )
```

可以证明是成立的，并可应用在替换中。

但是，表达式的含义只是简单的求值结果，这也是为什么子表达式可以相互独立地进行求值的原因。表达式的含义可以非常简单，比如数字3。而一条命令的含义是一个状态的转换或者具有类似复杂性的东西。想要理解一条命令，必须了解它对于机器状态所起的所有作用。

1.2 过程式程序设计语言中的表达式

自从Fortran以来, 程序设计语言到底进步了多少呢? 看一下欧几里得算法(辗转相除法), 这个算法是递归定义的, 用来计算两个自然数的最大公因子(Greatest Common Divisor, GCD):

$$\begin{aligned} \gcd(0, n) &= n \\ \gcd(m, n) &= \gcd(n \bmod m, m) \end{aligned} \quad \text{当 } m > 0 \text{ 时}$$

在Pascal这种过程式语言中, 很多人会将GCD编写成一个命令式程序:

```
function gcd(m, n: integer): integer;
  var prevm: integer;
begin
  while m <> 0 do
    begin prevm := m; m := n mod m; n := prevm end;
  gcd := n
end;
```

而下面是一个使用Standard ML写的函数式程序:

```
fun gcd(m, n) =
  if m=0 then n
  else gcd(n mod m, m);
```

命令式程序, 虽然是用所谓的“高级”语言写的, 但它并不比机器语言写的程序清楚、简单多少。它重复地更新三个量, 其中一个只是临时的存储区。想要证明它的确实现了欧几里得算法需要一套冗长乏味的弗洛伊德-霍尔(Floyd-Hoare)证明规则。相对地, 那个函数式版本的程序显然实现了欧几里得算法。

用Pascal也可以书写递归程序, 但那只是一个小的改善。递归的过程调用很难高效地实现。递归被引入程序设计30多年后, 仍被认为是应该避免使用的算法。历史上, 递归过程的正确性证明充满了令人遗憾的错误和复杂性。

Pascal的表达式并不满足通常的数学定律。一个优化编译器很可能想将 $f(z) + u/2$ 变形为 $u/2 + f(z)$, 但是, 如果函数 f 改变了 u 值的话, 这两个表达式就可能得不到相同的值。Pascal表达式的含义不仅涉及值也关系到状态。在所有现实的应用中, 都丧失了引用透明。

在纯的函数式语言中是没有状态的。表达式在机器精度的范围内(例如实数运算是近似的)满足通常的数学定律。纯函数式程序是可以利用Standard ML书写的, 不过, ML并不纯粹, 因为它有赋值语句和输入输出命令。那些风格“几乎”是函数式的ML程序员最好不要有引用透明的错觉。

1.3 存储管理

过程式语言中的表达式在Fortran之后就没什么进展了, 没跟上数据结构发展的步伐。假如我们有包含姓名、地址和其他信息的雇员记录, 我们并不能写出以记录为值的表达式, 或是从函数返回一个雇员记录, 即便有的语言允许这样做, 拷贝如此大的记录也是慢得令人无法忍受。

为了避免拷贝大的对象, 我们可以间接地引用它们。以记录作为返回值的函数可以为雇员记录分配存储空间, 然后返回它的存储地址。代替将记录来回拷贝, 我们只是拷贝了它的

地址。在雇员记录使用结束后，再来释放它的存储空间。以这种方式来使用的地址被称作引用 (reference) 或指针 (pointer)。

释放是这种方法的大问题。记录仍在使用的时候，程序可能就释放了它的存储空间，这时，当这个空间被再次分配时，它会同时被用作不同的目的。这之后，任何事情都可能发生，（可能是在很久以后）导致程序莫名其妙地崩溃。这是最危险的程序设计错误之一。

如果永远不释放存储空间，我们就会遇到没有存储空间的时候。那么，是不是要避免使用引用呢？然而，很多基本的数据结构（如链接表）是离不开引用的。

函数式语言以及其他一些语言是自动管理存储空间的。程序员并不用决定什么时候去释放一个记录的存储空间。运行时系统间歇性地扫描存储空间，标记出那些仍可以访问到的空间，然后将剩下的回收。这个操作称为垃圾收集 (garbage collection)，尽管称之为回收更合适。垃圾收集可能会很慢并需要额外的空间，但它是值得的。

具有垃圾收集功能的语言通常都是大量使用引用来作为数据的内部表示法。“返回”一个雇员记录的函数其实只是返回它的引用而已，但是，程序员不知道也不关心这一点。语言会因此获得更丰富的表达能力。程序员在摆脱了存储管理的杂事以后，工作会更有成效。

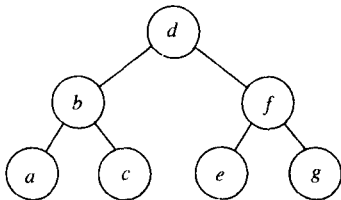
1.4 函数式语言的元素

函数式程序是跟值打交道的，而不是跟状态打交道的。它们的工具是表达式，而不是命令。那么怎样才能免去赋值、数组和循环呢？难道这个世界没有状态吗？这样的问题确实很有挑战性。不过，函数式程序员自有一套解决问题的技术。

表 (list) 和树 (tree)。数据集可以作为下面这样的表来处理：

$[a, b, c, d, e, \dots]$

表支持顺序访问：从左到右地进行扫描。这对于大部分应用来说已经足够了，甚至可以应用于排序和矩阵运算。更为灵活的方式就是把数据组织成树：



平衡树允许随机访问：可以很快速地到达任何一部分。理论上，树提供了和数组相同的效率；不过实践中，数组通常更快一些。在符号运算中，树扮演了非常重要的角色，比如，它在定理自动证明中可以表达那些逻辑项和公式。表和树都是通过引用来表示的，因此，运行时系统要带有垃圾收集器。

函数 (function)。表达式主要是由函数调用组成的。为了增强表达式的能力，必须取消对于函数的种种强迫限制。函数可以使用任何类型的参数，也可以返回任何类型的结果。如我们将看到的那样，“任意类型”包括函数本身，都可以像其他数据一样地使用，为了能这样使用，同样需要垃圾收集器。

递归 (recursion)。在函数式程序中，变量是通过外部传入（当一个函数被调用时）或者声明来获得值的。变量不能被再次更新，不过递归调用可以制造一系列变化的参数值。递归

要比迭代更容易理解，如果你不相信这点的话，尽可以参考上面的两个GCD程序。递归省掉了过程式语言里结构复杂的循环构造。[⊖]

模式匹配 (pattern-matching)。大多数函数式语言都允许函数通过模式匹配来分析它的参数。计算表中元素个数的函数在ML中是这样的：

```
fun length []      = 0
  | length (x::xs) = 1 + length xs;
```

我们立刻就看到了空表 ([]) 的长度是零，并且一个以元素 x 开头，后接子表 xs 的表的长度是子表 xs 的长度加上一。这里有一个等价的函数定义，它是用没有模式匹配的语言Lisp写的：

```
(define (length x)
  (if (null? x)
      0
      (+ 1 (length (cdr x)))))
```

ML的函数通常要考虑很多情况，遇到的模式要比 $x::xs$ 复杂得多。不用模式匹配来表达这样的函数是非常麻烦的。ML的编译器在内部做了模式匹配，要比程序员自己来做得好得多。

多态类型检测 (polymorphic type checking)。作为人，程序员经常会犯错误。使用数据结构并不存在的一部分，给一个函数提供过少的参数，以及混淆对象的引用和对象本身都是非常严重的错误：它们都有可能使程序崩溃。但幸运的是，如果一种语言强制使用类型作为律条的话，编译器是可以在程序运行前发现这些错误的。类型 (type) 将数据归类为整数、实数、表等，并且让我们可以保证以合理的方式使用它们。

有些程序员反对类型检测，因为这种检测可能会过于严格。在Pascal中，一个计算表长度的函数必须指定——一个完全无关的东西——表元素的类型。ML的长度函数对于所有的表都有效，因为ML的类型系统是多态的 (polymorphic)：它忽略了无关的数据的类型。我们的Lisp版本也同样可以不指定表的数据类型，因为Lisp根本就没有编译时的类型检测。Lisp比ML还要灵活，一个表中可以有不同类型的元素。这种自由的代价就是，本来可以自动捕获的错误需要花上几个小时才能发现。

高阶函数 (higher-order function)。函数本身也是可计算的值。甚至就连Fortran都允许一个函数作为另一个函数的参数传入，但是几乎没有过程式语言允许函数作为值来扮演数据结构所能扮演的一切角色。

高阶 (higher-order) 函数，或称为算子 (functional)，是一种操作于另一个 (或几个) 函数之上的函数。 map 算子，当应用于函数 f 上时，返回另一个函数，这个被返回的函数将

$[x_1, x_2, \dots, x_n]$ 变换成 $[f(x_1), f(x_2), \dots, f(x_n)]$

另一个高阶函数，当应用于函数 f 和一个值 e 时，返回

$f(x_1, f(x_2, \dots, f(x_n, e) \dots))$

如果 $e = 0$ 且 $f = +$ (是的，加法运算是一个函数) 的话，我们就得到了 x_1, \dots, x_n 的和，它是这样计算的

$x_1 + (x_2 + \dots + (x_n + 0) \dots)$

⊖ 递归也确实受到过批评。Backus (1978) 建议提供迭代原语来取代大多数函数定义中所使用的递归。但是，他的函数式程序设计风格并没有赶上发展的步伐。

如果 $e = 1$ 且 $f = \times$ 的话，我们就得到了它们的连乘积，它是这样计算的

$$x_1 \times (x_2 \times \cdots \times (x_n \times 1) \cdots)$$

其他的一些计算也可以通过选择适当的 f 和 e 来完成。

无穷数据结构 (infinite data structure)。像 $[1,2,3,\dots]$ 这样的无穷表也可以赋予计算上的意义。当碰到较为深入的问题时使用它们可以带来方便。无穷表是通过惰性求值 (lazy evaluation) 来处理的，这使得无穷表不会计算任何值或其中的部分值，除非这些值是获得最后结果所必需的。一个无穷表永远不会以完整的面目出现，可以把它看作是计算相继元素的过程。

自动定理证明系统中的搜索空间可以构成一棵无穷树，里面相连的结点则可以构成一个无穷表。不同的搜索策略产生出具有不同相连结点的表。这个 (抽象的无穷的) 表又可以被传到程序的其他部分，对于程序来说，表是如何产生的并不重要。

无穷表也可以表示输入输出序列。我们很多人都会在Unix操作系统的管道 (pipe) 中遇到这个概念。由管道连接而成的一系列进程组成了一个大的进程。每个进程都在消费就绪的输入，然后将输出通过管道传递给下一个进程。中间进程的输出从不会被完整地存储下来。这不但节省存储开销，更重要的是，它提供了清晰的组合进程的表示法。从数学的角度看，每个进程都是一个将输入变换成输出的函数，而进程链则是函数的复合。

输入和输出。和具有状态的外部世界通信，在函数式程序设计里则变得不太自然了。无穷表确实可以处理输入输出序列 (就像上面提到的)，但交互式的程序设计和进程间的通讯问题却是非常棘手的。为了解决这个问题，人们研究出很多种函数式的方法，单子 (monad) 是其中最有成效的一个 (Peyton Jones和Wadler, 1993)。而ML则只是简单地提供了输入输出命令，也就是说，ML在这个问题上放弃了函数式程序设计。

8

❶ 函数式语言的概况。主流的函数式语言采用了惰性求值、模式匹配和ML方式的多态类型。Miranda是一种优雅的语言，它出自David A. Turner (1990a)。Lazy ML是ML的一个带有惰性求值的变种，它的编译器可以产生高效的代码 (Augustsson和Johnsson, 1989)。Haskell是由研究员们组成的一个委员会设计的，作为一种通用的语言 (Hudak等, 1992)，已经被广泛地采用了。

John Backus (1978) 在一个大型的公开演讲中介绍了他的FP语言。FP提供了大量的高阶函数 (被称为“组合式”)，但程序员却不能定义新的高阶函数。Backus批评了程序设计语言和底层执行硬件的紧耦合，引进了冯·诺依曼瓶颈 (von Neumann bottleneck) 来描绘处理器和存储间的联系。许多意见认为函数式语言在并行的硬件上运行非常理想。Sisal就是被设计用来进行并行数值计算的，Cann (1992) 声称Sisal有时在性能上超过了Fortran。

许多函数式程序设计的实现技术，例如垃圾收集，都是源自Lisp (McCarthy等, 1962) 的。这种语言包含了底层的功能，若错误地应用就可能导致灾难性的后果。后来的变种，包括Scheme (Abelson和Sussman, 1985) 和Common Lisp，提供了高阶函数。虽然许多Lisp代码都是命令式的，但第一个函数式程序却是用Lisp写出来的。大多数ML的变种都包含一些命令式的功能，但是ML要比Lisp更有原则。ML有编译时的类型检测，并且只允许更新可变对象。

1.5 函数式程序设计的效率

典型的函数式程序都有庞大的运行系统和常驻编译程序。垃圾收集器可能需要统一的数据表示方式，这会占用额外的空间。函数式程序员有时会失去一些非常高效的数据结构，例如，数组、字符串和位向量。因此，函数式程序可能不如相应的C程序那样高效，尤其是在空间的需求上。

ML最适合大型的、复杂的应用程序。函数式程序设计中的类型检测、自动存储分配和其他优点可以让人体会到一个程序能否正确工作的区别。效率这时成为了次要的问题；另外，对于一个本来就有很大开销的程序，效率上的差别就没那么明显了。大多数函数式程序和相应的过程式程序差不多快，也许在最坏的情况下要慢五倍。

许多研究人员都怀疑效率的重要性，无疑是因为追求效率曾经导致了很多程序的崩溃。函数式程序员有时为了清晰宁可选择低效的算法，或者是想办法去丰富一种语言而不是想着怎样更好地实现它。更多的是这种态度，而不是什么技术上的原因，使得函数式程序设计给人留下了效率低下的印象。

现在，我们必须重新选择平衡点。函数式程序必须是高效的，否则没人会去用它。算法最终也是为了效率而设计的。两个数的最大公因子当然可以在所有可能的数里面去寻找而得出。这种穷尽的搜索算法非常清晰，但是毫无用处。欧几里得算法虽然牺牲了一些清晰性，却简单而快速。

求取GCD的穷举搜索算法是一个可执行描述（executable specification）的例子。一种程序设计的方法是：以此为起点，通过应用变换（transformation）来提高效率，并同时保持算法的正确性。最后，它可能会得到欧几里得算法。程序变换确实可以改善效率，不过我们应该小心的看待可执行描述。两个整数的最大公因子的定义是：可以同时整除这两个数的最大整数。这个描述里根本就没有提到什么搜索。穷举搜索算法并不是一个好的描述方法，因为它太复杂了。

函数式程序设计和逻辑式程序设计都属于声明式程序设计（declarative programming）。最理想的声明式程序设计就是不用我们写什么程序，只是声明我们的要求，然后计算机就会处理余下的工作。Hoare（1989c）曾经用最大公因子的例子探讨了这个理想，演示的结果表明这依然是一个梦想。更为现实的一个目标就是利用声明式程序设计来使得程序更加易懂。并且可以通过简单的数学论证来判断它们的正确性，而不用思考字节层面的问题。声明式程序设计依旧是程序设计，我们仍然必须编制出高效的代码。

这本书给出了帮助你判断哪里需要注意效率问题的具体建议。大多数自然的函数式定义同时也是相当高效的。一些ML编译器提供了运行评价功能（execution profiling），它可以测量每个函数的运行时间。那个花费时间最长的函数（从来不是你所认为的那个）将成为主要的改进对象。这种自底向上的优化可以产生显著的效果，虽然它不能揭示出全局低效的原因。对于程序设计来说，这些建议都是通用的，不论是函数式的、过程式的、面向对象的或者其他什么。

正确性无疑是首要的，清晰性通常次之，而效率则是第三位的。任何牺牲清晰性的程序都会更难于维护，除非这样做能显著地提升效率。明智地兼顾现实和原则，加上足够的耐心，就可能做出高效的程序。

❶ 函数式程序设计的应用。函数式程序设计主要应用在人工智能、形式方法、计算机辅助设计以及其他涉及符号运算的项目中。很多编译器都是用（也是为了）Standard ML（Appel, 1992）和Haskell（Peyton Jones, 1992）书写的。曾经有网络软件也是用ML书写的（Biagioni等, 1994），它演示了ML在系统程序设计方面的用途。一个重要的自然语言处理系统，叫做LOLITA，是用Haskell书写的（Smith等, 1994），作者采用函数式程序设计来处理这个复杂的系统。Hartel和Plasmeijer（1996）描述了六个主要的函数式程序设计项目，涉及各个方面的应用。Wadler和Gill（1995）曾经将这些实际应用汇编成表，其覆盖了很多的领域并且涉及了所有重要的函数式语言。

10

Standard ML概述

几乎所有成功的语言起初都是为了某种特殊的用途而设计的：Lisp是为了人工智能，Fortran是为了数值计算，Prolog是为了自然语言处理。相对地，那些为了通用目的设计的语言，例如那些“算法语言”Algol 60和Algol 68，其作为一种思想比作为一种实用的工具更为成功。

ML是为了自动定理证明而设计的。这并不是一个很广泛的领域，而且ML是专门为了编写其中一种定理证明机而设计的，目的非常明确！这个叫做爱丁堡LCF的（可计算函数的逻辑，Logic for Computable Function）定理证明机繁衍出了一堆后继者，它们都是用ML编写的。就像Lisp、Fortran和Prolog的许多应用已经远离了语言的设计初衷一样，ML也被用于各种各样的领域中。

1.6 Standard ML的演化

正如ML被称为证明策略编程的元语言（Meta Language）那样，它的设计者为了这个应用加入了必要的功能：

- 推导规则和证明方法都是要用函数来表示的，因此ML被赋予了高阶函数式程序设计的全部能力。
- 推导规则需要定义一个抽象类型：定理类型。强类型检测（像在Pascal里面的）会过分严格，因此ML采用了多态类型检测。
- 证明方法可能以非常复杂的方式组合。必须发现所有的失败，然后，才能尝试其他的方法。因此，ML拥有了抛出和捕获异常的功能。
- 如果一个定理证明机有漏洞的话，那么，它将毫无用处，因此ML设计得非常安全，绝不会破坏其运行环境。

为爱丁堡LCF设计的ML系统是非常缓慢的：首先程序被翻译成Lisp，然后进行解释执行。Luca Cardelli为自己的ML版本书写了高效的编译器，里面包括了丰富的声明和类型结构。在剑桥大学和INRIA，LCF的ML系统得到了扩充和性能上的改进。ML也影响了HOPE，这种纯函数式语言采用了多态性，并增加了递归类型定义和模式匹配。

11

Robin Milner曾试图将这些变种整合到Standard ML中去。很多人都参与到了其中。模块语言——最具复杂和创新的语言功能——是由David MacQueen设计，Milner和Mads Tofte优化

的。1987年，Milner因为他在Standard ML中的工作而获得了英国计算机界优秀技术大奖。第一批编译器是在剑桥和爱丁堡大学中研发的，不久，非常出色的Standard ML of New Jersey就问世了。

有几所大学都以教授Standard ML来作为学生的第一门程序设计语言。ML适合所有层次的学生，不论他们事先是否知道C、Basic、机器语言或是什么都不知道。使用ML，学生们可以学会如何使用数学思想去分析问题，打破那些低级语言培养的坏习惯。大量的计算可以在几行之内表达清楚。初学者特别喜欢类型检测器可以检测出常见的错误，而且系统绝不会崩溃。

在1.5节中我们提到了Standard ML在网络和编译器构造等方面的应用。然而，定理证明仍旧是ML的一个最重要的应用领域，下面将会看到。

❶ 进一步的阅读。Gordon等（1979）描述了LCF。Landin（1966）讨论了ISWIM语言，ML最初就是以这个语言为基础的。正式的Standard ML定义已经出版成书（Milner等，1990），并且有单独的一卷评论（Milner和Tofte，1990）^①。

Standard ML并没有取代所有的变种。特别是法国人，他们另辟蹊径。他们的CAML语言（Cousineau和Huet，1990）广泛地提供了类似的功能，不过保留了原先ISWIM的传统语法。它在语言设计实验方面是有用的，并且它的扩展超过了Standard ML，包括了惰性数据结构（lazy data structure）和动态类型。CAML Light是一个简单的字节码解释器，它很适合小型的计算机系统。采用惰性求值的ML变种仍然存在，前面也提到过了。HOPE也继续被使用和传授着（Bailey，1990）。

1.7 ML的自动定理证明传统

自动定理证明和函数式程序设计手拉手地成长。最早的一批函数式程序之一就是一个简单的定理证明机（McCarthy等，1962）。早在20世纪70年代，当一些研究人员还在奇怪着函数式程序设计到底有什么用的时候，爱丁堡LCF已经使用它来工作了。

完全自动的定理证明几乎是不可能有的：对于大多数逻辑来说，没有已知的自动方法。另一个明显的自动定理证明的替代品——证明检测，很快就让人难以接受。大多数证明都涉及冗长而反复的规则组合。

爱丁堡LCF代表了一种新型的定理证明机，其中自动化的层次完全由使用者决定。它基本上是一个可编程的证明检测机。使用者可以使用ML——Meta Language（元语言）——来书写证明的过程，而不用输入重复的命令。ML程序可以在对象语言的表达式上操作，这种对象语言就叫做Scott可计算函数逻辑。

爱丁堡LCF引入了将逻辑表达作为一个关于定理的抽象类型的思想。每条公理都是一个基本的定理，而推导规则就是从定理到定理的函数。类型检测保证了定理只能由公理和规则产生。将推导规则应用于已知的定理，一条规则接一条规则地向前进行，便构造出了证明。

策略（tactic）允许使用更自然的反向证明形式。每个策略都是一个从目标到子目标的函数，这个函数需要判断是否存在一条反向的推导规则。策略实际上是返回了这个推导规则

① 根据作者的勘误表：Milner的书已经有了1997年的新版。——译者注

(作为一个函数): 因此, 策略是高阶函数。

策略算子 (tactical) 提供了将简单策略组合成复杂策略的控制结构。得出的策略可以进一步组合成更为复杂的策略, 这样的策略可能一步就进行了几百条基本的推导。策略算子比策略更为“高阶”。高阶函数的新用途出现在重写技术等其他方面。

❶ 进一步的阅读。自动定理证明的起源是人工智能中的一个任务。后来的研究将它应用到了推理任务上, 例如制定计划 (Rich和Knight, 1991)。程序验证的目的是证明软件的正确性。然而, 作为新领域的硬件验证却更成功, Graham (1992) 描述了一个相当复杂的VLSI芯片的验证, 同时也对其他工作进行了评述。

爱丁堡LCF的分支包括了使用高阶逻辑的HOL88 (Gordon和Melham, 1993), 以及支持构造性论证的Nuprl (Constable等, 1986)。

其他近期的系统则采用Standard ML。LAMBDA是一个硬件合成工具, 它用于设计电路并同步地利用高阶逻辑来证明电路的正确性。ALF是一个构造性类型理论的证明编辑器 (Magnusson和Nordström, 1994)。

1.8 新标准库

ML的定义中描述了一个包含标准声明的小型库, 里面包括了关于数、字符串和表的操作。很多人都觉得这个库不够用。例如, 里面没有将字符串"3.14"转换成实数的功能。在人们将ML用到系统程序设计和其他没有预见到的领域时, 这个库的缺点变得更加明显。一个由多个编译器编写小组组成的委员会起草了新的ML标准库 (Gansner和Reppy, 1996)。在写这本书的时候它还在开发中, 不过我们已经知道了它的基本框架。^①

这个库需要对ML本身做出一些小的改动, 它引进了字符类型, 与长度为一的字符串加以区分。它允许内部表示不同的多个数值类型同时存在, 因此也保持了它们各自的精度, 这改变了我们对待一些数值函数的做法。

库是由ML的模块组成的。很多函数都是ML结构 (structure) 的成员, 这些结构的内容是通过ML签名 (signature) 来描述的。函数并不能单由它的名字来调用, 而必须带上它所属结构的名称; 例如, 实数的符号函数是`Real.sign`而不是`sign`。很多函数不止出现在一个结构中, 例如, 库也提供`Int.sign`。当我们后面讨论到模块时, 库可以帮助我们了解其关键概念。下面总结了库的主要组成部分, 以及相关的结构:

- 表和序偶表的操作属于结构`List`和`ListPair`, 它们中的一些内容会在稍后章节讲述。
- 整数操作属于结构`Int`。整数可能有几种精度。这里面可能包括了通常硬件支持的整数 (结构`FixedInt`), 非常高效但是大小有限。也可能包括无限精度的整数 (结构`IntInf`), 它是某些任务所必需的。
- 实数操作属于结构`Real`, 不过像`sqrt`、`sin`和`cos`这样的函数则属于`Math`。实数也可能会有几种精度。像`Real32`或`Real64`这样名字的结构表明了实数所用的二进制位数。
- 无符号整数的运算也是允许的。这个包括了通常只在低级语言里面才会有的位操作, 例如逻辑“与”。ML的版本是安全的, 它不允许二进制位被任意地转换成其他类型。支持

^① 根据作者的勘误表: 在翻译这本书的时候, 基本库仍在开发中, 不过应该在2005年完成。——译者注

此类运算的结构具有类似*Word8*这样的名字。

- 多种形式的数组。这里面包括了通常命令式语言里的可变更数组（结构*Array*），以及不可变数组（结构*Vector*）。因为后者是不可更新的，所以适合函数式程序设计。它们的初值是通过计算得到的，这些计算的开销往往是很大的，因此不适合重复地进行。
- 关于字符和字符串的操作是属于结构*Char*和*String*的。而关于某个类型和这个类型的文字表示之间的转换则放在了该类型的相关结构中，例如*Int*。
- 输入和输出有几种支持方式。主要的两种是传输文本的文本I/O；和传输任意字节流的二进制I/O。相关的结构是*TextIO*和*BinIO*。
- 操作系统的原语放在结构*OS*中。它们都与文件、目录和进程有关。其中也可能提供了相当数量的其他操作系统和输入输出服务。
- 日期和时间的操作，包括处理器时间的测量功能都在结构*Date*、*Time*和*Timer*中。
- 库中不同部分都需要的那些声明则集中在结构*General*中。

有许多其他的软件包和工具包，虽然不在库中，却也很广泛的存在。最终的运行环境甚至可以支持大多数苛求的项目。

1.9 ML和工作中的程序员

软件因不可靠而声名狼藉。Wiener（1993）阐述了无数因软件失败而导致的生命损失、商业危机和其他灾难的案例。软件产品不但不提供使用保证，而且都附有免责条款。我们能通过使用ML代替C进行程序设计来避免这些失败吗？当然不能，不过这可能是一个正确的努力方向。

问题的一部分是对于安全性的损害非常普遍。检测数组和引用是否正确使用的代价是昂贵的，但检测可以使得错误在造成严重损害之前就被发现。C. A. R. Hoare曾经说道：

……这样做是荒谬的：我们对结果无所谓的调试运行做大量精细的检测，而对错误可能导致沉重代价甚至灾难的正式运行去掉了这些检测。试想，一个航海爱好者在干地上训练时穿着救生衣，一旦真正下到海里便脱掉它。会怎么样呢？（Hoare，1989b，第198页）

这段话在1973年的一个演讲中第一次被提到后，几乎没有被重视过。典型的编译器都会省略掉检测，除非是有特别的命令去包含它们。C语言是非常不安全的：像它的数组仅仅是个存储地址，检测它们的使用正确与否根本不现实。标准C函数库包含了许多可能会冲垮内存的函数，它们接受一个存储区域作为参数，却不知道这个区域的大小！这样的后果就是：Unix操作系统有很多的安全漏洞。互联网蠕虫利用了这点，制造了大量的网络破坏（Spafford，1989）。

ML从许多方面支持可靠的软件开发。编译器不允许省略检测。Appel（1993）提到了ML的安全性、自动存储分配和编译时类型检测，这些因素合在一起能消除一些主要的错误，并保证其他错误的早期发现。Appel同意函数式程序设计是有价值的观点，即便是在大型项目中。

此外，ML是正式定义的。Milner等（1990）的定义并不是第一个正式的程序设计语言定义，但它是第一个编译器作者能看懂的定义。[⊖] 由于常见的歧义性没有了，因此编译器可以在很大的范围内达到一致。新的标准库更加强了这种一致性。一个程序不论用哪个编译器编

⊖ 这可能要感谢程序设计语言理论的近期成果。ML的定义是结构化操作语义的一个范例（Hennessy，1990）。

译，都应该有相同的表现，ML已经非常接近这个理想了。

其中一个关键的长处就是ML的模块系统。系统的组成部分，不论多大，都可以将描述和编码分开。每个组件都可以提供已经描述好的服务，并且外界不能篡改。组件可以将其他的组件作为参数，并且分别进行编译。这样的组件可以用很多种方法组合，配置出不同的系统。

从软件工程的角度看，ML是一种适合大型系统的优秀语言。它的模块让程序员可以分组工作并重用组件。它的类型和总体的安全增强了可靠性。它的异常功能让程序可以响应失败。比较了ML和C，Appel承认ML程序需要非常多的空间，但运行的速度却是可以接受的。软件开发人员也有几种商业化的编译器可以选择。

我们不能指望短期内ML程序可以运行在电子手表上。但是，对于大型的应用，可靠性和程序员的成效是基本的要素。是不是C的年代就要结束了呢？

第2章 名字、函数和类型

大多数的函数式语言都是交互式的。如果你输入一个表达式，编译器会立刻进行运算并给出结果。交互式非常有意思，它可以即时反馈，可以让你通过易于管理的小部分代码来一点点地开发程序。

我们可以输入一个表达式，并以分号结束……

```
2+2;
```

……然后ML回应

```
> 4 : int
```

这里我们看到一些后面都要用到的书写风格。大多数ML系统都会在等待输入的时候显示一个提示符；这里，输入是以打字机式的字体显示的，回应则显示为斜体字：

```
> on a line like this.
```

简单地说，ML就是一个计算器。它有整数，像上面那样，以及实数。ML可以进行简单的算术运算……

```
3.2 - 2.3;
```

```
> 0.9 : real
```

……还有平方根运算：

```
Math.sqrt 2.0;
```

```
> 1.414213562 : real
```

强调一下，打入ML的任何东西都要以分号(;)结束。ML打印出了值和类型。注意，*real*是实数类型，而*int*则是整数类型。

交互式的程序设计对于过程式的语言来说更为困难，因为它们太啰唆了。一个独立的程序作为一次的输入来说实在是太长了。

17

本章提要

本章介绍Standard ML和函数式程序设计。基本的概念包括声明、简单数据类型、记录类型、递归函数和多态性。虽然这些材料是以Standard ML来表现的，但是它叙述了一般性的原理。

本章分为以下几节：

- 值的声明。用基本的例子来介绍值和函数的声明。
- 数、字符串和真值。内置的类型*int*、*real*、*char*、*string*和*bool*支持算术运算、文字和逻辑操作。
- 序偶、元组和记录。序偶和元组使函数可以有多个参数和结果。
- 表达式的求值。严格求值和惰性求值的区别不仅仅是效率，也和表达式的确切含义相关。

- 书写递归函数。举几个实际的例子来讲解递归的应用。
- 局部声明。使用`let`和`local`，可以在有限制的作用域内声明名字。
- 模块系统初步。签名和结构是通过对于算术运算操作的泛化开发来介绍的。
- 多态类型检测。介绍了多态性的原理，包括类型推导和多态函数。

值的声明

声明 (declaration) 就是赋予某个东西一个名字。ML里有许多东西可以赋予名字：值、类型、签名、结构和函子。在ML中大多数的名字代表了值，例如数、字符串以及函数。虽然在ML里函数也是一种值，不过它们有自己的声明语法。

2.1 命名常量

任何重要的值都是可以命名的，不论其重要性是广泛的（像常数 π ）或是暂时的（像最近一次计算的结果）。举个小例子，假设想计算一小时里面有多少秒，可以先用名字`seconds`代表60。

18

```
val seconds = 60;
```

值的声明是以ML的关键字`val`开始，并以分号结束的。本书中的名字通常用斜体字表示。ML重复显示一遍这个名字，以及它的值和类型：

```
> val seconds = 60 : int
```

我们再来定义每小时多少分钟和每天多少小时这些常量：

```
val minutes = 60;
> val minutes = 60 : int
val hours = 24;
> val hours = 24 : int
```

现在这些名字就可以用在表达式里面了：

```
seconds*minutes*hours;
> 86400 : int
```

如果你像上面这样在（运行环境的）顶层输入一个表达式，ML把求得的值存入名字`it`。通过引用`it`，你可以将该值用在后面的计算中：^①

```
it div 24;
> 3600 : int
```

名字`it`里总是放着最近一次在顶层输入的表达式值。任何之前`it`的值都被丢弃了。想要保留`it`的值，就要声明一个永久的名字：

```
val secsinhour = it;
> val secsinhour = 3600 : int
```

顺便提一下，名字里面可以包含下划线，这样看起来更清楚：

```
val secs_in_hour = seconds*minutes;
> val secs_in_hour = 3600 : int
```

① SML/NJ和其他一些编译器在回应计算结果的时候会显示诸如`val it = 3600 : int`，不过这里仍采用原书写法。作者并没表示示范代码属于某个特定的编译器。——译者注

为了演示一下实数，我们通过公式 $area = \pi r^2$ 来计算半径为 r 的圆面积：

```
val pi = 3.14159;
> val pi = 3.14159 : real
val r = 2.0;
> val r = 2.0 : real
val area = pi * r * r;
> val area = 12.56636 : real
```

2.2 声明函数

计算圆面积的公式可以像下面这样写成ML的函数：

```
fun area (r) = pi*r*r;
```

19

函数声明以关键字`fun`开始，而`area`是函数的名字， r 是形式参数（formal parameter），`pi*r*r`则是函数体（body）。函数体里面引用了 r 和上面声明的常量 pi 。

因为函数在ML里面也是值，函数声明就是值声明的一种形式，所以ML也打印了值和类型：

```
> val area = fn : real -> real
```

这个类型用正规的数学记法是 $real \rightarrow real$ ，表示`area`以一个实数作为参数并返回另一个实数。函数的值表示为`fn`。和大多数函数式语言一样，ML里的函数也是一种抽象值：它们的内部结构是隐藏的。

让我们调用一下函数，重复上面做过的面积计算：

```
area(2.0);
> 12.56636 : real
```

我们再试试别的参数，注意参数外面的括号是可选的：

```
area 1.0;
> 3.14159 : real
```

函数声明里括号也是可选的。下面的`area`的函数声明和之前的等价：[⊖]

```
fun area r = pi*r*r;
```

函数应用的求值过程会在下面进行详细地讨论。

注释。程序员们常常认为他们的作品太清晰了，不需要更多的描述。逻辑的清晰对其他人来说并不明显，除非程序注释得很恰当。注释可以描述声明的目的，可以给出文字上的参考，或是解释晦涩的问题。当然，注释一定要是正确而不过时的。

ML里面的注释由`(*`开始，以`*)`结束，并且可以跨越数行。注释甚至可以嵌套。它们几乎可以被插入在任何地方：

⊖ 在几乎所有的函数式程序设计中，参数外面的括号习惯上是不用的，这一点和过程式语言很不一样。函数调用在这里也习惯称为函数应用（function application），意思是将函数应用到实际参数上，而调用一词会给人留下程序运行这样的过程式语言的印象。因为函数应用是函数式语言的主要操作，所以它采用最省事的写法：`fa`。后面讲到参数的柯里化时也会看到这种记法的好处。函数式语言因此也称为应用式（applicative）语言，不过后来这个词被大多用作与声明式（declarative）同义，而与命令式（imperative）相对了。——译者注

```
fun area r =      (*半径为r的圆的面积*)
    pi*r*r;
```

函数式的程序员不应该觉得可以免去书写注释。人们还曾经宣称Pascal是自成文档的语言呢。

20

名字的重新声明。值的名字称作变量 (variable)。和命令式语言里面的变量不同, 这里的变量是不能更新的。不过一个名字可以被重新用作其他的用途。如果一个名字被再次声明, 那么新的含义将在之后被采用, 但不会影响现有的对这个名字的使用。让我们重新声明常量`pi`:

```
val pi = 0.0;
> val pi = 0.0 : real
```

我们可以看到`area`仍然使用`pi`原先的值:

```
area(1.0);
> 3.14159 : real
```

到了这个阶段, 几个变量都有了值。包括`seconds`、`minutes`、`area`和`pi`, 以及库里面提供的内置操作。在所有地方都可以看到的这些绑定[⊖]被称作环境 (environment)。函数`area`引用了一个较早的环境, 在那里`pi`代表3.14159。这要感谢名字的永久性 (称作静态绑定, static binding), 重新声明一个名字不会损坏系统、库或是你的程序。



修改你的程序。由于静态绑定, 重新声明你的程序所调用的函数可能没有任何作用。当修改程序的时候, 要保证把整个文件再编译一遍。大程序应该分成模块, 将在第7章详述。在修改过的模块重新编译以后, 整个程序仅需要重新连接。

2.3 Standard ML中的标识符

一个字母名字 (alphabetic name) 必须以字母开始, 后面则可以跟随任意数目的字母、数字、下划线 (`_`) 或撇号 (`'`), 通常也叫做单引号。例如:

```
x    UB40    Hamlet_Prince_of_Denmark    h''3_H
```

大小写是有区别的, 所以`q`和`Q`不同。允许撇号是因为ML是数学家们设计的, 他们喜欢把变量叫做`x`、`x'`、`x''`。在选择名字的时候, 要确定避免使用ML的关键字:

```
abstype and andalso as case datatype do
else end eqtype exception fn fun functor
handle if in include infix infixr let local
nonfix of op open orelse raise rec
sharing sig signature struct structure
then type val where while with withtype
```

特别是注意那些短的关键字: `as`、`fn`、`if`、`in`、`of`、`op`。

21

ML也允许用符号名字 (symbolic name)。这些名字由下面的字符组成:

```
! % & $ # + - * / : < = > ? @ \ ~ ` ^ |
```

由这些字符组成的名字可以任意长:

⊖ 绑定 (binding): 名字 (变量) 和值的结合。——译者注

```
---->    $^$^$^$    !!?@**??!!    :-|==>->#
```

某些特殊字符组成的串是为ML的语法所保留的，它们不能被用作符号名字：

```
: | = => -> # :>
```

凡是字母名字可以出现的地方，符号名字也可以出现：

```
val +--+ = 1415;
> val +--+ = 1415 : int
```

名字更为正式的称呼是标识符 (identifier)。标识符可以同时表示值、结构、签名、函子和记录域。

练习 2.1 在你的计算机系统上学习怎样开始和结束一个ML会话。然后学习怎样让ML编译器从文件里面读入声明，一个典型的命令就是 `use "myfile"`。

数、字符串和真值

ML里最简单的值就是整数和实数、字符串和字符以及布尔值或真值。这一节介绍这些类型和它们的常量以及基本操作。

2.4 算术运算

ML区分整数 (类型 *int*) 和实数 (类型 *real*)。整数算术是精确的 (有些ML系统里面支持无限的精度)，而实数算术只能像计算机里面的浮点数硬件那样精确。

整数。一个整数常量是一串数字，有可能以一个负号 (~) 开始。例如：

```
0    ~23    01234    ~85601435654678    ,
```

整数运算包括了加法 (+)、减法 (-)、乘法 (*)、除法 (*div*) 和取模 (*mod*)。这些都是中缀运算符，它们遵循常规的优先级：因此在

```
((m*n)*k) - (m div j)) + j
```

22

这个式子里面，所有的括号都是可以去掉的，而含义不变。

实数。一个实数常量包含一个小数点或者E记法，或者两者都有。例如：

```
0.01    2.718281828    ~1.2E12    7E~5
```

结尾En的意思是“乘以10的n次方”。负的指数部分是以一元减号 (~) 开始的。这样的话 `123.4E~2` 就表示 `1.234`。

负实数以一元减号 (~) 开始。实数的中缀运算符包括加法 (+)、减法 (-)、乘法 (*) 和除法 (/)。函数应用比中缀运算符优先级更高。例如，`area a + b` 等价于 `(area a) + b`，而不是 `area (a + b)`。



一元加号和减号。一元减号是一个否定号 (~)。不要将它和减号 (-) 搞混！

ML里面没有一元的加号。+和-都不能出现在实数的指数部分。

类型约束。ML可以根据表达式里面用到的函数和常量的类型推导出大多数表达式的类型。不过某些内置的函数是被重载 (overloaded) 了的，它们不止有一个含义。例如，+和*对于整数和实数中都有定义。重载函数的类型必须由上下文来确定，偶尔必须显式地指出。

例如, ML不能确定这个平方的函数是给整数用的, 还是实数用的, 因此拒绝接受这样的声明。

```
fun square x = x*x;
> Error- Unable to resolve overloading for *
```

假设函数是针对实数的。我们可以在几个地方插入类型`real`。

我们可以指定参数的类型:

```
fun square(x : real) = x*x;
> val square = fn : real -> real
```

也可以指定结果的类型:


```
fun square x : real = x*x;
> val square = fn : real -> real
```


同样, 我们可以指定函数体的类型:

```
fun square x = x*x : real;
> val square = fn : real -> real
```

23

类型约束也可以出现在函数体内部, 实际上几乎任何地方都可以。

 **默认重载。**标准库引入了默认重载的记法, 编译器可以通过选择类型`int`来解决`square`的歧义。在这种情况下使用类型约束也是有益的, 这增加了清晰性。默认重载的初衷是允许不同精度的数同时存在。例如, 除非1.23的精度可以从上下文确定, 否则就将以默认精度的实数来表示。在写这本书的时候, 还没有遇到过不同精度的数的表示, 但是这方面的考虑是必要的。

 **算术运算和标准库。**标准库包括了很多不同精度的整数和实数的函数。结构`Int`包括了类似`abs` (绝对值)、`min`、`max`和`sign`的函数。这里是一些例子:

```
Int.abs ~4;
> 4 : int
Int.min(7, Int.sign 12);
> 1 : int
```

结构`Real`包含了类似的函数, 比如`abs`和`sign`, 以及用于在整数和实数之间转换的函数。调用`real(i)`将`i`转换为等值的实数。调用`round(r)`将`r`转换为值最接近的整数。其他实数到整数的转换还包括`floor`、`ceil`和`trunc`。每当整数和实数出现在同一个表达式里时就需要使用转换函数了。

结构`Math`包含了有关实数的更高级的数学函数, 比如`sqr`、`sin`、`cos`、`atan` (反正切)、`exp`和`ln` (自然对数)。每个函数都是以一个实数为参数, 并返回实数类型的结果。

练习 2.2 一个Lisp黑客说过: “由于整数是实数的子集, 它们之间的区别完全是人为的, 是那些硬件设计者强加于我们的。ML应该简单得只提供数, 就像Lisp那样, 自动选择合适的整数或实数。”你同意吗? 出于什么样的考虑呢?

练习 2.3 下面哪些函数需要类型约束?

```
fun double(n) = 2*n;
fun f u = Math.sin(u) / u;
fun g k = ~ k * k;
```

2.5 字符串和字符

消息和其他文本都是字符组成的串。它们的类型是`string`。字符串常量是用双引号括起来的：

```
"How now! a rat? Dead, for a ducat, dead!";
> "How now! a rat? Dead, for a ducat, dead!" : string
```

24

连接操作符 (^) 将两个字符串首尾相接：

```
"Fair " ^ "Ophelia";
> "Fair Ophelia" : string
```

内置函数`size`返回一个字符串里面的字符个数。这里`it`是指`"Fair Ophelia"`：

```
size (it);
> 12 : int
```

里面的空格当然也算。空字符串不包含任何字符，`size("")`是0。

下面的函数给名字加上一个贵族头衔：

```
fun title(name) = "The Duke of " ^ name;
> val title = fn : string -> string
title "York";
> "The Duke of York" : string
```

特殊字符。由一个反斜线开始的转义序列 (escape sequence) 可以将某些特殊字符插入到字符串中。这里列出一些：

- `\n` 插入一个换行符。
- `\t` 插入一个制表符。
- `\"` 插入一个双引号。
- `\\` 插入一个反斜线。
- `\` 之后紧跟一个换行和其他空白字符，然后再跟一个`\`则不插入任何字符，不过在换行之后续写同一个字符串。

下面是一个包含换行符的字符串：

```
"This above all:\nto thine own self be true\n";
```

字符类型。就像数字3和集合{3}的区别那样，一个字符不同于只包含一个字符的字符串。字符具有`char`类型。它的常量形如`#s`，`s`则是只有一个字符的字符串常量。下面定义了一个字母、一个空格和一个特殊字符：

```
#"a"      #" "      #"\\n"
```

函数`ord`和`chr`将字符和字符码互相转换。大多数实现都是使用ASCII字符集，如果 $0 \leq k < 255$ ，那么`chr(k)`返回一个以`k`为字符码的字符。相应地，`ord(c)`返回字符`c`的字符码。我们可以利用这些来将0到9之间的数转换成`#"0"`到`#"9"`之间的字符：


25

```
fun digit i = chr(i + ord #"0");
> val digit = fn : int -> char
```

函数`str`和`String.sub`将字符和字符串互相转换。如果`c`是一个字符的话, `str(c)`则是相对应的字符串。相应地, 如果`s`是一个字符串的话, 那么`String.sub(s, n)`则返回`s`里面的第`n`个字符, 从零开始计数。让我们试着用另一种方式表达函数`digit`:

```
fun digit i = String.sub("0123456789", i);
> val digit = fn : int -> char
str (digit 5);
> "5" : string
```

第二个`digit`的定义比第一个更可取, 因为它不依赖于字符的编码方式。

 字符串、字符和标准库。结构`String`包含了很多关于字符串的操作。结构`Char`提供了诸如`isDigit`、`isAlpha`之类的判定字符类别的函数。子串(substring)是字符串里面的连续的字符子序列, 结构`Substring`提供了提取和处理它们的操作。

ML的定义(Milner等, 1990)里面只有字符串类型。标准库引入了字符类型。标准库同时修改了一些内置函数的类型, 比如`ord`和`chr`, 它们以前是对只有一个字符的字符串进行操作的。

练习 2.4 对于`digit`的两个版本, 你觉得调用`digit 1`和`digit 10`都会得到什么响应? 在上机实验之前试着预测一下。

2.6 真值和条件表达式

为了通过分情来定义函数, 也就是说函数的结果取决于测试的结果, 我们引入条件表达式。[⊖]测试就是一个类型为`bool`的表达式`E`, 取值可以是`true`(真)和`false`(假)。测试的结果用于在两个表达式`E1`或`E2`中选择其一。条件表达式

```
if E then E1 else E2
```

的值有两种可能, 当`E`等于`true`时, 为`E1`, 或当`E`等于`false`时, 为`E2`。`else`部分必须存在。

最简单的测试是关系运算:

- 小于 (<)
- 大于 (>)
- 小于或等于 (<=)
- 大于或等于 (>=)

这些关系在整数和实数上都有定义, 它们也可以按字母顺序测试字符串和字符的大小。这样一来, 关系运算就被重载了, 有可能需要类型约束。相等(=)和不相等(<>)对于大多数类型都可以进行测试。

例如, 函数`sign`计算一个整数的符号(1、0或-1)。它包括两个条件表达式和一条注释。

```
fun sign(n) =
  if n>0 then 1
  else if n=0 then 0
```

[⊖] 因为Standard ML的表达式可以改变状态, 条件表达式可以像过程式语言的`if`命令一样使用。

```

    else (*n<0*)      ~1;
  > val sign = fn : int ->int

```

测试可以通过ML的布尔运算进行组合:

- 逻辑或 (orelse)
- 逻辑与 (andalso)
- 逻辑非 (函数not)

返回布尔值的函数也叫做谓词 (predicate)。下面是一个测试作为其参数的字符是不是小写字母的谓词:

```

fun isLower c = #"a" <= c andalso c <= #"z";
> val isLower = fn : char -> bool

```

当对条件表达式求值时, 要么then表达式被求值, 要么else表达式被求值, 但它们从不会同时被求值。布尔操作符andalso和orelse不同于普通的函数: 第二个操作数只是在需要的时候才会被求值。它们的名字反映了这种顺序性的行为。

练习 2.5 假设 d 是一个整数且 m 是一个字符串。书写一个ML的布尔表达式, 当且仅当 d 和 m 组成一个有效的日期时为真: 比如25和"October"。假设不在闰年里。

序偶、元组和记录

在数学里, 几个值集成在一起通常也看作是一个值。二维空间中的一个向量是一个有序的实数对。一个关于两个向量 \vec{v}_1 和 \vec{v}_2 的语句也可以换作一个关于四个实数的语句, 甚至那些实数本身也可以再分割成更小的部分, 但是在高的层次思考会容易一点。书写 $\vec{v}_1 + \vec{v}_2$ 要比书写 $(x_1 + x_2, y_1 + y_2)$ 更省事。

27

日期是个更为普通的例子。像25 October 1415这样的日期由三个值组成。把它作为一个整体看, 则是一个形如(day, month, year)的三元组。这个基本概念经过了相当长的时间才出现在程序设计语言中, 并且仅有几种能恰当地处理它。

Standard ML提供了序偶 (二元组)、三元组、四元组等。在 $n > 2$ 时, n 个值的有序集成被称为 n 元组, 或简称元组 (tuple)。分量为 x_1, x_2, \dots, x_n 的元组书写成 (x_1, x_2, \dots, x_n) 。这样的值是由形如 (E_1, E_2, \dots, E_n) 的表达式建立的。通过元组, 函数可以有多个参数和多个结果。

ML元组的分量本身也可以是元组或任何其他值。例如, 一段时间可以通过日期的序偶来表达, 而不管日期是如何表示的。另外, 嵌套的序偶也可以表示 n 元组。(在Classic ML这个最原始的变种中, $(x_1, \dots, x_{n-1}, x_n)$ 只不过是 $(x_1, \dots, (x_{n-1}, x_n) \dots)$ 的简写。)

ML记录 (record) 的分量是由标识符确定的, 而不是由位置确定的。一个有20个分量的记录打印出来要占很大的地方, 不过它要比20元组容易管理。

2.7 向量: 序偶的例子

我们来书写几个关于向量的例子。想要试验序偶的语法, 可以输入向量(2.5, -1.2):

```

(2.5, ~1.2);
> (2.5, ~1.2) : real * real

```

这个向量类型的数学写法是 $real \times real$, 也就是实数序偶的类型。向量是ML里的值, 也可以赋予名字。下面声明零向量以及其他两个向量 a 和 b :


```

val zerovec = (0.0, 0.0);
> val zerovec = (0.0, 0.0) : real * real
val a = (1.5, 6.8);
> val a = (1.5, 6.8) : real * real
val b = (3.6, 0.9);
> val b = (3.6, 0.9) : real * real

```

许多关于向量的函数都是在其分量上操作的。 (x, y) 的长度是 $\sqrt{x^2 + y^2}$ ，而它的反向量是 $(-x, -y)$ 。在ML里面编写这些函数，只要简单地将参数写成一个模式：

```

fun lengthvec (x,y) = Math.sqrt(x*x + y*y);
> val lengthvec = fn : real * real -> real

```

函数`lengthvec`以 x 和 y 组成的序偶作为参数。它具有类型 $real \times real \rightarrow real$ ：它的参数是实数序偶，而结果则是另一个实数。^① 下面的 a 是刚才声明过的实数序偶。

```

lengthvec a;
> 6.963476143 : real
lengthvec (1.0, 1.0);
> 1.414213562 : real

```

函数`negvec`相对于原点 $(0,0)$ 反转一个向量。

```

fun negvec (x,y) : real*real = (~x, ~y);
> val negvec = fn : real * real -> real * real

```

这个函数的类型是 $real \times real \rightarrow real \times real$ ：给定一个实数序偶返回另一个序偶。因为负号 $(~)$ 是被重载的，所以类型约束 $real \times real$ 是必需的。

我们反转一些向量，并给 b 的反向量命名：

```

negvec (1.0, 1.0);
> (~1.0, ~1.0) : real * real
val bn = negvec(b);
> val bn = (~3.6, ~0.9) : real * real

```

向量可以是函数的参数和结果，也可以命名。总的来说，它们拥有和ML中内置类型（比如整数）完全一样的权利。我们甚至可以声明一个向量类型：

```

type vec = real*real;
> type vec

```

现在`vec`就是 $real \times real$ 的缩写。它也只是个缩写：所用实数序偶的类型都是`vec`，不管该序偶是不是为了表示一个向量。我们可以使用`vec`作为类型约束。

2.8 多参数和多结果的函数

下面是计算两个实数平均值的函数。

```

fun average(x,y) = (x+y)/2.0;
> val average = fn : (real * real) -> real

```

对一个向量做这件事是很奇怪的，不过`average`对任何两个（实）数都起作用：

```

average(3.1,3.3);
> 3.2 : real

```

^① 函数`Math.sqrt`只对实数定义，因此约束了被重载的运算符，使其必须是实数类型的。

一个在序偶上定义的函数其实就是一个具有两个参数的函数： $lengthvec(x, y)$ 和 $average(x, y)$ 操作在实数 x 和 y 上。是否将 (x, y) 看作向量完全取决于我们自己。类似地， $negvec$ 有两个参数，并返回两个结果。

严格地说，每个ML函数只有一个参数和一个返回结果。利用元组，函数可以有任意多个参数，以及返回任意多个结果。

由于一个元组的分量本身也可以是一个元组，所以两个向量可以凑成一对：

```
((2.0, 3.5), zerovec);
> ((2.0, 3.5), (0.0, 0.0)) : (real*real) * (real*real)
```

两个向量 (x_1, y_1) 和 (x_2, y_2) 的和向量是 $(x_1 + x_2, y_1 + y_2)$ 。在ML里，这个函数以一对向量为参数。它的参数模式是一对序偶：

```
fun addvec ((x1,y1), (x2,y2)) : vec = (x1+x2, y1+y2);
> val addvec = fn : (real*real) * (real*real) -> vec
```

类型 vec 首次出现了，约束了加法，使其必须作用在实数上。ML给出了 $addvec$ 类型

$$((\text{real} \times \text{real}) \times (\text{real} \times \text{real})) \rightarrow \text{vec}$$

这等价于更为简练的写法 $(\text{vec} \times \text{vec}) \rightarrow \text{vec}$ 。ML系统不可能将所有的 $\text{real} \times \text{real}$ 都简化成 vec 。

再来看看 $addvec$ 的参数模式。我们可以等价地将这个函数的参数看作

- 一个参数：实数序偶的序偶。
- 两个参数：每个都是一个实数序偶。
- 四个参数：全都是实数，莫名其妙地分成两组。

下面我们将向量 $(8.9, 4.4)$ 和 b 加起来，然后把结果和另一个向量再加起来。留意一下，函数的结果类型是 vec 。

```
addvec((8.9, 4.4), b);
> (12.5, 5.3) : vec
addvec(it, (0.1, 0.2));
> (12.6, 5.5) : vec
```

向量的减法涉及到分量的相减，但也可以通过其他向量操作来表示：

```
fun subvec(v1,v2) = addvec(v1, negvec v2);
> val subvec = fn : (real*real) * (real*real) -> vec
```

变量 $v1$ 和 $v2$ 都在实数序偶的范围内取值。

```
subvec(a,b);
> (~2.1, 5.9) : vec
```

两个向量的距离就是两个向量差的长度：

```
fun distance(v1,v2) = lengthvec(subvec(v1,v2));
> val distance = fn : (real*real) * (real*real) -> real
```

由于 $distance$ 不会分开使用 $v1$ 和 $v2$ ，所以它可以简化为：

```
fun distance pairv = lengthvec(subvec pairv);
```

变量 $pairv$ 在向量序偶的范围内取值。这个版本看上去可能有点奇怪，不过确实和之前的版本等价。 a 到 b 有多远呢？

```
distance(a,b);
> 6.262587325 : real
```

最后一个例子将说明序偶的分量可以是不同类型的：这里是一个实数和一个向量。缩放一个向量就是将两个分量乘以同一个常量。

```
fun scalevec (r, (x,y)) : vec = (r*x, r*y);
> val scalevec = fn : real * (real*real) -> vec
```

同样，类型约束`vec`保证乘法是作用在实数上的。函数`scalevec`以一个实数和一个向量为参数，并返回一个向量。

```
scalevec(2.0, a);
> (3.0, 13.6) : vec
scalevec(2.0, it);
> (6.0, 27.2) : vec
```

选择元组的分量。在一个模式上，比如 (x, y) ，定义的函数通过模式变量 x 和 y 来引用参数里面的分量。`val`声明也可以将值和模式匹配：每一个模式中的变量将指向相应的分量。

31

下面我们把`scalevec`看作是返回两个结果的函数，这两个结果分别命名为`xc`和`yc`。

```
val (xc,yc) = scalevec(4.0, a);
> val xc = 6.0 : real
> val yc = 27.2 : real
```

`val`声明里的模式可以和函数定义里的参数模式一样复杂。在下面这个特意构造的例子里，一个序偶的序偶被分解成四个部分，每个部分都给予命名。

```
val ((x1,y1), (x2,y2)) = (addvec(a,b), subvec(a,b));
> val x1 = 5.1 : real
> val y1 = 7.7 : real
> val x2 = ~2.1 : real
> val y2 = 5.9 : real
```

零元组和单元类型`unit`。之前我们只考虑 $n \geq 2$ 的 n 元组。也存在零元组，写作`()`并读做“单元”(unity)，它没有分量。它的用途在于，当没有数据需要传输的时候提供一个占位符。零元组是`unit`类型里唯一的一个值。

`unit`类型经常用于ML中的过程式程序设计。过程就是返回值类型为`unit`类型的“函数”。过程的调用是为了看到它的作用——而不是得到它的值，反正值总是`()`。例如，一些ML系统提供一个类型为`string \rightarrow unit`的函数`use`。调用`use "myfile"`的作用是将文件“myfile”里面的定义读入ML。

参数类型是`unit`的函数在调用时不能给函数体传递任何信息。调用这样的函数只不过是函数体进行求值。在第5章，这样的函数被用在延时求值上，可以实现无穷表的程序设计。

练习 2.6 书写一个函数来判断一天之内的某个时间，形如 $(hour, minute, AM或PM)$ ，是否在另一个时间之前。例如，判断 $(11, 59, "AM")$ 是否在 $(1, 15, "PM")$ 之前。

练习 2.7 古老的英国钱币是12便士为一先令，且20先令为一英镑。写一些函数，利用三元组 $(pounds, shillings, pence)$ 来加减两个钱数。

2.9 记录

记录是一种其分量——称作域(field)——具有标签的元组。一个 n 元组的每个分量是由

它所在的位置，从1到n，来标识的，而记录里的域却可以以任意顺序出现。调换一个元组的分量通常会导致错误。如果雇员是由元组(*name*, *age*, *salary*)来表示的话，("Jones", 25, 15300)和("Jones", 15300, 25)的区别就大了。但是，记录

```
{name="Jones", age=25, salary=15300}
```

和

```
{name="Jones", salary=15300, age=25}
```

是一样的。记录是由花括号括起来的，每个域都形如*label* = *expression*。

记录适用于多个分量的情况。让我们记录一些英国国王的五种基本事实，并注意ML的回应：

```
val henryV =
  {name   = "Henry V",
   born   = 1387,
   crowned = 1413,
   died   = 1422,
   quote  = "Bid them achieve me and then sell my bones"};
> val henryV =
>   {born = 1387,
>     died = 1422,
>     name = "Henry V",
>     quote = "Bid them achieve me and then sell my bones",
>     crowned = 1413}
> : {born: int,
>     died: int,
>     name: string,
>     quote: string,
>     crowned: int}
```

ML把域重新调整为一种标准的顺序，忽略了输入的顺序。记录的类型中列出了每一个域，形如*label*: *type*，并用花括号括起来。下面是另外两个国王的记录：

```
val henryVI =
  {name   = "Henry VI",
   born   = 1421,
   crowned = 1422,
   died   = 1471,
   quote  = "Weep, wretched man, \
\ I'll aid thee tear for tear"};

val richardIII =
  {name   = "Richard III",
   born   = 1452,
   crowned = 1483,
   died   = 1485,
   quote  = "Plots have I laid..."};
```

*henryVI*的*quote*延伸到了两行，使用了反斜线-换行-反斜线的转义序列。

记录模式。由形如*label* = *variable*的域构成的记录模式赋予每一个变量相应的标签值。如果我们不需要所有的域，那么可以在其他域所在的地方写三个点(...)。下面我们来取得Henry V记录的两个域，把它们叫做*nameV*和*bornV*：

```
val {name=nameV, born=bornV, ...} = henryV;
> val bornV = 1387 : int
> val nameV = "Henry V" : string
```

通常，我们想要打开一个记录，以便直接看到里面的域。可以用模式`label = label`来指定每一个域，使变量和标签同名。这种描述可以简单地缩写为`label`。现在打开记录Richard III:

```
val {name, born, died, quote, crowned} = richardIII;
> val crowned = 1483 : int
> val born = 1452 : int
> val died = 1485 : int
> val quote = "Plots have I laid..." : string
> val name = "Richard III" : string
```

要想省略一些域，就像上面那样写 (...). 现在`quote`代表了Richard III的名言。显然这只适合一次一个国王。

记录域的选择。选择符`#label`可以从记录里取得给定的`label`的值。

```
#quote richardIII;
> "Plots have I laid..." : string
#died henryV - #born henryV;
> 35 : int
```

不同的记录类型可以包含同样的域标签。雇员和国王都有`name`，"Jones"或是"Henry V"。上面给出的三个国王记录属于同一记录类型，因为它们具有相同数目的域，域的标签和类型也都相同。

这是另外一个不同记录类型包含相同域标签的例子： n 元组 (x_1, x_2, \dots, x_n) 就是一个使用数

34

作为域标签的记录的简写：

$$\{1 = x_1, 2 = x_2, \dots, n = x_n\}$$

没错，域标签可以是正整数！有必要了解这个Standard ML中的奇特事实，其唯一的理由就是：使用选择操作符`#k`可以提取 n 元组里面的第 k 个分量的值。因此`#1`选择第一个分量，而`#2`选择第二个分量，如果还有第三个分量的话，用`#3`选择它，依此类推：

```
#2 ("a", "b", 3, false);
> "b" : string
```

⚠ 部分记录描述。仅选择一些域而忽略另一些域是不能完全确定记录类型的；一个函数只能定义在完整的记录类型之上。比如，不能定义一个函数让它接受所有具有`born`和`died`域的记录，必须指定所有的域标识（通常是利用类型约束）。这种限制使得ML的记录很有效率但不方便。这种限制同样适用于记录模式和域选择操作`#label`。Ohori (1995) 曾经在一个ML的变种中定义和实现了弹性记录。

声明记录类型。让我们声明国王的记录类型。这种缩写在函数的类型约束中很有用。

```
type king = {name    : string,
              born    : int,
              crowned : int,
              died    : int,
              quote   : string};
> type king
```

我们现在可以声明一个关于类型`king`的函数来得到国王的寿命：

```
fun lifetime(k: king) = #died k - #born k;
> val lifetime = fn : king -> int
```

利用模式，`lifetime`也可以这样声明：

```
fun lifetime(({born,died,...}: king) = died - born;
val lifetime = fn:king -> int
```

不论哪种声明方式，类型约束都是必需的。否则，ML会提示“需要一个固定的记录类型”。

```
lifetime henryV;
> 35 : int
lifetime richardIII;
> 33 : int
```

35

练习 2.8 下面的函数定义是否需要类型约束？它的类型是什么？

```
fun lifetime(({name,born,crowned,died,quote}) = died - born;
```

练习 2.9 探讨一下域选择符`#born`和下面的函数有没有区别？有的话区别在哪里？

```
fun born_at({born}) = born;
```

2.10 中缀操作符

中缀操作符 (infix operator) 是一个写在它的两个参数中间的函数。我们使用中缀运算符是为了和数学记法看齐。设想没有中缀操作符，我们只有将 $2+2=4$ 写成 $=(+(2,2),4)$ 了。大多数的函数式语言都允许程序员声明自己的中缀操作符。

我们这就声明一个中缀操作符`xor`代表“异或”运算。首先使用ML的`infix`指令：

```
infix xor;
```

我们现在必须写`p xor q`，而不是`xor(p, q)`：

```
fun (p xor q) = (p orelse q) andalso not (p andalso q);
> val xor = fn : (bool * bool) -> bool
```

`xor`函数以一个布尔序偶为参数，并返回布尔结果。

```
true xor false xor true;
> false : bool
```

如果有区别的话，中缀的状态只是影响到一个函数的语法，而不是它的值。通常，名字首先被指定成中缀，然后才定义它的值。

中缀的优先级。大多数人认为 $m \times n + i/j$ 的意思是 $(m \times n) + (i/j)$ ，给予乘法和除法高于加法的优先级。类似地， $i - j - k$ 的意思是 $(i - j) - k$ ，这是因为减法运算符是左结合的。一条ML的`infix`指令可以声明一个从0到9的优先级。默认的优先级是0，也就是最低级的。指令`infix`规定操作符是左结合的，而`infixr`则规定右结合。

为了演示中缀操作符，下面的函数将几个字符串放在括号里面。操作符`plus`的优先级是6（也就是ML里加号的优先级），它构造一个包含加号的字符串。

```
infix 6 plus;
fun (a plus b) = "(" ^ a ^ "+" ^ b ^ ";";
> val plus = fn : string * string -> string
```

36

仔细观察, *plus*是左结合的:

```
"1" plus "2" plus "3";
> "((1+2)+3)" : string
```

类似地, *times*的优先级是7 (如同ML里的乘法), 它构造了一个包含乘号的字符串。

```
infix 7 times;
fun (a times b) = "(" ^ a ^ "*" ^ b ^ ")";
> val times = fn : string * string -> string
"m" times "n" times "3" plus "i" plus "j" times "k";
> "(((m*n)*3)+i)+(j*k))" : string
```

传统的幂函数用操作符*pow*表示, 它比乘法的优先级高, 并且是右结合的。函数用#符号来表示幂。(ML里面没有幂运算符。)


```
infixr 8 pow;
fun (a pow b) = "(" ^ a ^ "#" ^ b ^ ")";
> val pow = fn : string * string -> string
"m" times "i" pow "j" pow "2" times "n";
> "(m*(i#(j#2)))*n" : string
```

很多中缀操作符都使用符号名字。让++作为向量加法的操作符:

```
infix ++;
fun ((x1,y1) ++ (x2,y2)) : vec = (x1+x2, y1+y2);
> val ++ = fn : (real*real) * (real*real) -> vec
```

它和*addvec*的作用一样, 只不过使用了中缀记法:

```
b ++ (0.1,0.2) ++ (20.0, 30.0);
> (23.7, 31.1) : vec
```

 隔开符号名字。符号名字如果连在一起用会产生误会。下面, ML将+~读作一个符号名字, 然后抱怨这个名字没定义:

```
1+~3;
> Unknown name +~
```

两个符号名字必须用空格或其他字符隔开:

```
1+ ~3;
> ~2 : int
```

37 将中缀作为函数。有时候, 中缀操作符必须像一个普通的函数一样使用。在ML里面, 关键字*op*覆盖了中缀状态: 如果⊕是一个中缀操作符的话, 那么*op*⊕就是普通的函数写法, 它可以像通常形式那样被应用到一个序偶上。

```
> op++ ((2.5,0.0), (0.1,2.5));
(2.6, 2.5) : real * real
op^ ("Mont", "joy");
> "Montjoy" : string
```

中缀状态也可以被取消。如果⊕是一个中缀操作符的话, 那么指令*nonfix* ⊕让它回到普通函数的记法。后面的*infix*指令又可以将⊕变成一个中缀操作符。

这里, 我们取消ML中乘法操作符的中缀状态。此时再试图使用通常的乘法就会产生错误, 因为我们不能将3用作函数。然而, 现在*可以被应用作普通的函数了:

```

nonfix *;
3*2;
> Error: Type conflict...
*(3,2);
> 6 : int

```

`nonfix`指令是为了交互式的语法开发而设计的，可以试验操作符不同的优先级和结合方式。随便改变已有操作符的中缀状态将导致混乱。

表达式的求值

命令式的程序指定命令来更新机器的状态。运行期间，状态每秒改变几百万次。它的结构也发生变化：局部变量不断地被创建和销毁。即使程序有一个与硬件细节无关的数学含义，这个含义也超出了程序员的理解力。公理语义和指称语义的定义只对少数专家有意义。程序员只是依赖调试工具和他们的直觉来修改程序。

函数式程序设计的目的是想给每个程序一个直接的数学含义。它简化了程序在我们脑海里执行的样子，因为没有状态的改变。执行只是将一个表达式简化成它的值，将相等的东西进行替换。对大多数函数定义的理解仅涉及初等数学。

当应用一个函数的时候，比如 $f(E)$ ，必须把参数 E 提供给 f 的函数体。如果一个表达式涉及了多个函数调用，那么必须根据一些求值规则来选择其中一个。ML 采用的求值规则是传值调用 (call-by-value) 或称为严格 (strict) 求值，而多数纯函数式语言则采用传需调用 (call-by-need) 或称为惰性 (lazy) 求值。

38

每种求值规则都有自己的派别。为了进行比较，我们来看两个简单的函数。平方函数 `sqr` 将参数使用了两次：

```

fun sqr(x) : int = x*x;
> val sqr = fn : int -> int

```

常函数 `zero` 忽略了它的参数而直接返回 0：

```

fun zero(x : int) = 0;
> val zero = fn : int -> int

```

当一个函数被调用时，函数体里的形式参数会被实际参数所替换。求值规则在什么时候对实际参数求值、进行几遍求值这些方面上是有所区别的。形式参数指示了在函数体的什么地方进行替换。此外，形式参数的名字并没有其他的意义，在函数定义之外也没有任何意义。

2.11 ML 中的求值：传值调用

假设表达式是由常量、变量、函数调用和条件表达式 (`if-then-else`) 构成的。常量具有显式的值；变量在环境中绑定到它的值；所以求值只是处理函数调用和条件表达式。ML 的求值规则是基于一个简单的想法。

为了求得 $f(E)$ 的值，首先对表达式 E 进行求值。

这个值替换了 f 函数体内的形式参数，然后函数就可以进一步求值了。模式匹配增加了一点点复杂性，如果 f 声明为：

```

fun f (x,y,z) = body

```


那么就用 E 值的相应部分去替换模式变量 x 、 y 和 z 。(一种现实的实现方法就是不作任何替换,而是将形式参数绑定到函数的局部环境中。)

看一下ML是怎样计算 $\text{sqr}(\text{sqr}(\text{sqr}(2)))$ 的。在三个函数调用中,只有最里面的那个参数有值。因此 $\text{sqr}(\text{sqr}(\text{sqr}(2)))$ 可以简化为 $\text{sqr}(\text{sqr}(2 \times 2))$ 。现在必须对乘法求值,得到了 $\text{sqr}(\text{sqr}(4))$ 。对内层调用求值得到了 $\text{sqr}(4 \times 4)$,依此类推。简化可以写作 $\text{sqr}(\text{sqr}(4)) \Rightarrow \text{sqr}(4 \times 4)$ 。完整的计算是这样的:

$$\begin{aligned}\text{sqr}(\text{sqr}(\text{sqr}(2))) &\Rightarrow \text{sqr}(\text{sqr}(2 \times 2)) \\ &\Rightarrow \text{sqr}(\text{sqr}(4)) \\ &\Rightarrow \text{sqr}(4 \times 4) \\ &\Rightarrow \text{sqr}(16) \\ &\Rightarrow 16 \times 16 \\ &\Rightarrow 256\end{aligned}$$

现在再看一下 $\text{zero}(\text{sqr}(\text{sqr}(\text{sqr}(2))))$ 。 zero 的参数是上面表达式的求值结果。求值是进行了,但是值又被忽略掉了:

$$\begin{aligned}\text{zero}(\text{sqr}(\text{sqr}(\text{sqr}(2)))) &\Rightarrow \text{zero}(\text{sqr}(\text{sqr}(2 \times 2))) \\ &\vdots \\ &\Rightarrow \text{zero}(256) \\ &\Rightarrow 0\end{aligned}$$

多浪费啊!像 zero 这样的函数是不多见的,但是经常会有一些函数,它们的结果并不依赖于所有的参数。

ML的求值之所以被称为传值调用,是因为总是将函数参数的值传进函数。不难看出,这是我们通常用纸笔进行手算所采取的步骤。几乎所有的程序设计语言都采用它。不过我们也需要看看那种只用一步就将 $\text{zero}(\text{sqr}(\text{sqr}(\text{sqr}(2))))$ 简化到0的求值方法。在讨论这个问题之前,必须要先看看递归。

2.12 传值调用下的递归函数

阶乘函数是递归的标准例子。它包括一个基本情形: $n = 0$, 此时求值过程停止。

```
fun fact n =
  if n=0 then 1 else n * fact(n-1);
> val fact = fn : int -> int
fact 7;
> 5040 : int
fact 35;
> 10333147966386144929666651337523200000000 : int
```

ML像下面那样对 $\text{fact}(4)$ 求值。实际参数4取代了函数体内的 n , 得到

```
if 4 = 0 then 1 else 4 * fact(4 - 1)
```

由于 $4 = 0$ 不成立, 条件表达式简化为 $4 \times \text{fact}(4 - 1)$ 。然后对 $4 - 1$ 求值, 整个表达式简化为 $4 \times \text{fact}(3)$ 。图2-1总结了求值过程。条件表达式没出现在内, 它们除了 $n = 0$ 时都差不多, $n = 0$ 时

条件表达式的值是1。

```

fact(4) ⇒ 4 × fact(4 - 1)
        ⇒ 4 × fact(3)
        ⇒ 4 × (3 × fact(3 - 1))
        ⇒ 4 × (3 × fact(2))
        ⇒ 4 × (3 × (2 × fact(2 - 1)))
        ⇒ 4 × (3 × (2 × fact(1)))
        ⇒ 4 × (3 × (2 × (1 × fact(1 - 1))))
        ⇒ 4 × (3 × (2 × (1 × fact(0))))
        ⇒ 4 × (3 × (2 × (1 × 1)))
        ⇒ 4 × (3 × (2 × 1))
        ⇒ 4 × (3 × 2)
        ⇒ 4 × 6
        ⇒ 24

```

图2-1 *fact*(4)的求值

fact(4)的计算完全是按照阶乘的数学定义进行的： $0! = 1$ ，而当 $n > 0$ 时， $n! = n \times (n - 1)!$ 。过程式语言中的递归过程的执行能这么简洁地表达吗？

迭代函数。*fact*(4)的求值有点奇怪。当递归展开的时候，越来越多的数等待着被乘。乘法迟迟不能进行，直到*fact*(0)，这时则要计算 $4 \times (3 \times (2 \times (1 \times 1)))$ 。这个笔算过程显示*fact*浪费了空间。

通过思考应该怎样来计算阶乘，可以得到一个更有效的函数版本。根据乘法结合律，每个乘法都是可以立即进行的：

$$4 \times (3 \times \text{fact}(2)) = (4 \times 3) \times \text{fact}(2) = 12 \times \text{fact}(2)$$

计算机是不会自动应用这个定律的，除非我们要它这么做。函数*facti*在*p*里面保留了一个不断变化的积，它的初值是1：

```

fun facti (n,p) =
  if n=0 then p else facti(n-1, n*p);
> val facti = fn : int * int -> int

```

比较图2-2所示的*facti*(4, 1)的求值过程和*fact*(4)的求值过程，前者一直保持着较小的中间表达式，每个乘法都可以立即进行，并且所需要的存储空间的大小是固定的。这个求值过程是迭代的 (iterative)，也叫做尾递归的 (tail recursive)。在6.3节中，我们会通过建立定理*facti*(*n*, *p*) = $n! \times p$ 来证明*facti*给出的结果是对的。

好的编译器可以检测出迭代形式的递归并高效执行。递归调用*facti*(*n* - 1, *n* × *p*)的结果不需进一步计算而直接作为*facti*(*n*, *p*)的结果。这样的尾调用 (tail call) 可以优化执行：将新值赋予参数*n*和*p*，然后直接跳回函数开始，避免了正式函数调用的开销。在函数*fact*中的递归调用不是尾调用，因为它返回的值要进行进一步地运算，也就是还要乘以*n*。

很多函数都可以通过增加一个参数来变成迭代的，就像*facti*中的*p*。有时迭代函数运行起

来要快得多。有时将函数变成迭代的是防止存储空间溢出的唯一办法。然而，给每个递归函数都增加一个参数是个坏习惯，它导致了难看的、费解的代码，甚至可能运行得更慢。

$$\begin{aligned}
 \text{facti}(4, 1) &\Rightarrow \text{facti}(4 - 1, 4 \times 1) \\
 &\Rightarrow \text{facti}(3, 4) \\
 &\Rightarrow \text{facti}(3 - 1, 3 \times 4) \\
 &\Rightarrow \text{facti}(2, 12) \\
 &\Rightarrow \text{facti}(2 - 1, 2 \times 12) \\
 &\Rightarrow \text{facti}(1, 24) \\
 &\Rightarrow \text{facti}(1 - 1, 1 \times 24) \\
 &\Rightarrow \text{facti}(0, 24) \\
 &\Rightarrow 24
 \end{aligned}$$

图2-2 $\text{facti}(4, 1)$ 的求值

条件表达式的特殊作用。条件表达式允许进行分情定义。回想一下阶乘函数是怎样定义的：

$$0! = 1$$

$$n! = n \times (n-1)!$$

当 $n > 0$ 时

这些等式确定了所有 $n > 0$ 的整数的 $n!$ 。忽略了第二个等式的条件 $n > 0$ 会导致矛盾：

$$1 = 0! = 0 \times (-1)! = 0$$

类似地，在条件表达式

$\text{if } E \text{ then } E_1 \text{ else } E_2$

中，ML仅当 E 为真的时候才会计算 E_1 ，而仅当 E 为假的时候才会计算 E_2 。

由于是传值调用，所以不可能存在一个ML的函数 cond 可以像条件表达式那样计算 $\text{cond}(E, E_1, E_2)$ 。我们可以尝试声明一个，并用它来编写阶乘函数：

```

fun cond(p, x, y) : int = if p then x else y;
> val cond = fn : bool * int * int -> int
fun badf n = cond(n=0, 1, n*badf(n-1));
> val badf = fn : int -> int

```

这可能看上去不错，不过每一个 badf 的调用都不会停下来。观察一下 $\text{badf}(0)$ 的计算过程：

$$\begin{aligned}
 \text{badf}(0) &\Rightarrow \text{cond}(\text{true}, 1, 0 \times \text{badf}(-1)) \\
 &\Rightarrow \text{cond}(\text{true}, 1, 0 \times \text{cond}(\text{false}, 1, -1 \times \text{badf}(-2))) \\
 &\vdots
 \end{aligned}$$

虽然 cond 永远不会需要所有的三个参数，但是传值调用规则却要对它们全部求值。这样一来，递归就不会结束。

条件的与和或。ML的中缀布尔操作符 andalso 和 orelse 并不是函数，只代表某类条件表达式。

表达式 $E_1 \text{ andalso } E_2$ 是下式的简写

if E_1 then E_2 else *false*

表达式 E_1 orelse E_2 是下式的简写

if E_1 then *true* else E_2

这两个操作符完成了布尔运算与和或，但是它们仅当需要时才会对 E_2 求值。如果它们是函数的话，传值调用规则会对两个参数都求值。其他ML中缀操作符则的确是函数。

andalso和orelse的顺序求值方式使得它们很适合于表达递归谓词（返回布尔值的函数）。函数powoftwo测试一个数是不是二的整数次幂：

```
fun even n = (n mod 2 = 0);
> val even = fn : int -> bool
fun powoftwo n = (n=1) orelse
                  (even(n) andalso powoftwo(n div 2));
> val powoftwo = fn : int -> bool
```

你可能已经预料到powoftwo是由条件表达式定义的，没错，是通过orelse和andalso来定义的。求值在结果一出来时就立刻停止了：

```
powoftwo(6) ⇒ (6 = 1) orelse (even(6) andalso ...)
              ⇒ even(6) andalso powoftwo(6 div 2)
              ⇒ powoftwo(3)
              ⇒ (3 = 1) orelse (even(3) andalso ...)
              ⇒ even(3) andalso powoftwo(3 div 2)
              ⇒ false
```

练习 2.10 写出powoftwo(8)的化简步骤。

练习 2.11 powoftwo是不是一个迭代函数？

2.13 传需调用或惰性求值

传值调用规则积聚了许多的不满意度。它多余地计算了 $zero(E)$ 中的 E ，也多余地计算了 $cond(E, E_1, E_2)$ 中的 E_1 或 E_2 。条件表达式和类似的操作不能作为函数。虽然ML提供了关键字andalso和orelse，但是我们没办法定义类似的东西。

能否把参数作为表达式而不是值传进函数呢？总的思路是这样的：

为了计算 $f(E)$ 的值，直接将 E 替换进 f 的函数体。

然后，计算所得到的表达式的值。

这就是传名调用（call-by-name）规则。它将 $zero(sqr(sqr(sqr(2))))$ 立即简化到0。但是对于 $sqr(sqr(sqr(2)))$ 却很糟糕，它重复使用了参数 $sqr(sqr(2))$ 。这个“简化”的结果是

$$sqr(sqr(2)) \times sqr(sqr(2))$$

之所以这样，是因为 $sqr(x) = x \times x$ 。

乘法和其他算术运算一样需要特别处理。它必须应用到值上，而不是应用在表达式上：它是严格（strict）函数的一个例子。为了计算 $E_1 \times E_2$ ，必须首先计算 E_1 和 E_2 。

让我们继续上面的平方求值。由于最外面的函数是 \times ，是个严格函数（参数不是值则不

能直接计算), 规则选择了先进行最左边的 sqr 调用。它的参数同样是重复的:

$$(sqr(2) \times sqr(2)) \times sqr(sqr(2))$$

完整的计算就像下面这样:

$$\begin{aligned} sqr(sqr(sqr(2))) &\Rightarrow sqr(sqr(2)) \times sqr(sqr(2)) \\ &\Rightarrow (sqr(2) \times sqr(2)) \times sqr(sqr(2)) \\ &\Rightarrow ((2 \times 2) \times sqr(2)) \times sqr(sqr(2)) \\ &\Rightarrow (4 \times sqr(2)) \times sqr(sqr(2)) \\ &\Rightarrow (4 \times (2 \times 2)) \times sqr(sqr(2)) \\ &\vdots \end{aligned}$$

会得到结果吗? 最后是会的。但是传名调用不会是我们所期望的求值规则。

传需调用 (call-by-need) 规则 (惰性求值) 类似传名调用, 不过保证了对每一个参数最多求一次值。它不直接将一个表达式替换进函数体, 而是在参数出现的地方用指针连接到表达式上。如果参数指向的表达式曾被求值, 那么这个参数出现的地方都共享这个值。指针结构构成了关于函数和参数的有向图。当图的一部分被求过值后, 图会被结果值所更新。这称为图归约 (graph reduction)。

图2-3示意了一个图归约过程。每一步都将一个 $sqr(E)$ 替换成 $E \times E$, 这里两个 E 是共享的。没有多余的重复: 总共只有三个乘法计算。我们似乎得到了两个世界里最好的, 对于 $zero(E)$ 来说也可以立即简化到0。但是对图的维护所需的代价是很高的。

45

$cond(E, E_1, E_2)$ 中的惰性求值效果和条件表达式一样, 但先决条件是元组 (E, E_1, E_2) 本身也是惰性求值的。这方面的细节是很巧妙的: 元组形式必须被看成是一个函数。对于像 (E, E_1, E_2) 这样的数据结构进行部分求值的思想——也就是只对 E_1 和 E_2 其中之一求值, 可以引出无穷表的概念。

严格求值和惰性求值的比较。传需调用做到了最少的求值。它似乎是获得高效的途径。但是它所需的维护开销也很大。它的实现仅在David Turner (1979) 将图归约应用到组合子 (combinator) 之后才真正变得可行。他使用了 λ -演算的某些晦涩的特性来开发新的编译技术, 研究员们仍在不断地进行完善。每种新的技术都有它的卫道士: 有些人说惰性求值才是真正的方向、真理和曙光。那为什么Standard ML不采用它呢?

即使 E 的计算不能终止, 惰性求值也会告诉我们 $zero(E) = 0$ 。这可是在数学传统面前耍刀了: 一个表达式是有意义的, 当且仅当它的每个部分都是有意义的。Alonzo Church, λ -演算的发明者, 更喜欢它的一个变体 (λI -演算), 这个变体是禁止像 $zero$ 这样的常函数的。

无穷数据结构使得数学论证复杂化了。要完全理解惰性求值除了需要知道 λ -演算之外, 还要懂一些论域理论。程序的输出不再是简单的一个值了, 而是一个部分求值的表达式。这些概念学起来都不简单, 而且很多都只不过是手段而已。如果只能借助求值的手段来思考的话, 我们比过程式的程序员也好不了多少。

效率方面也存在很多问题。有时惰性求值节省了相当多的空间; 有时它也会浪费空间。回想一下, $facti$ 在严格求值下比 $fact$ 要更有效率, 每个乘法都是立即进行的。而 $facti(n, p)$ 的惰性求值会立即对 n 求值 (这是为了测试 $n = 0$), 但对 p 则不然。这样, 乘法就积累起来了, 我们也就浪费了空间 (见图2-4)。

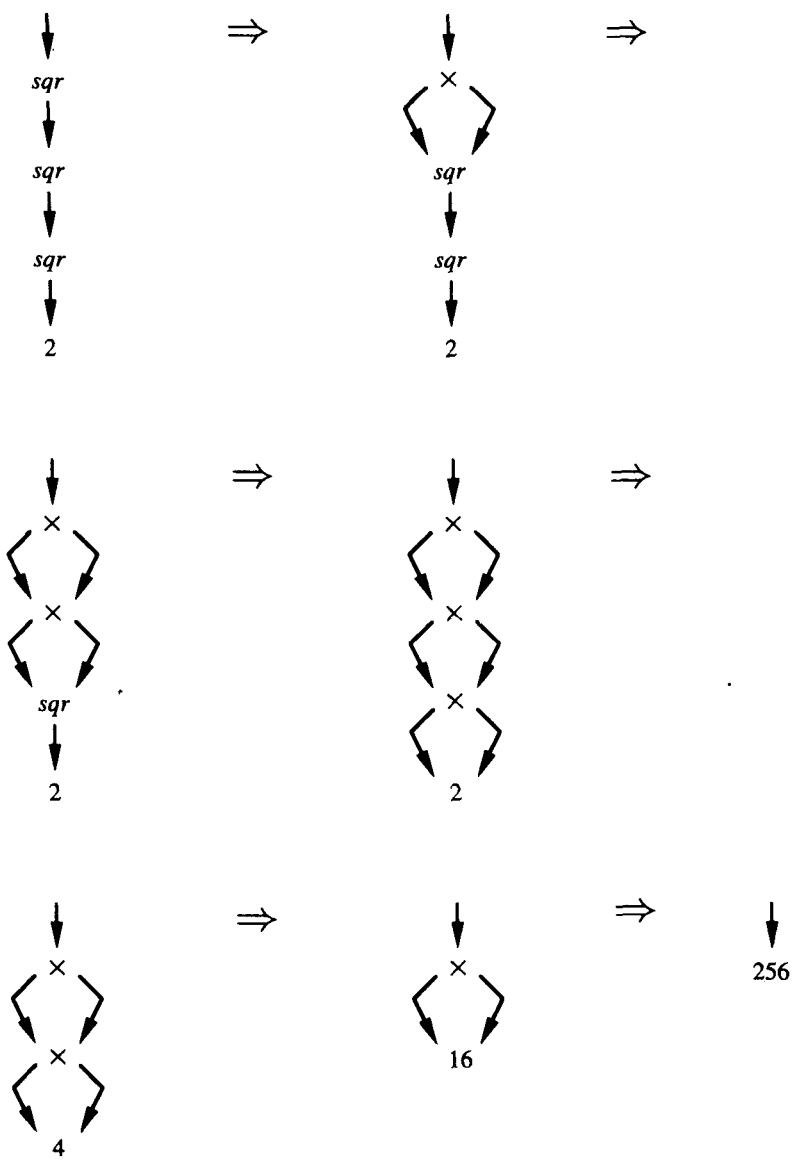


图2-3 $\text{sqr}(\text{sqr}(\text{sqr}(2)))$ 的图归约

$$\begin{aligned} \text{facti}(4, 1) &\Rightarrow \text{facti}(4 - 1, 4 \times 1) \\ &\Rightarrow \text{facti}(3 - 1, 3 \times (4 \times 1)) \\ &\Rightarrow \text{facti}(2 - 1, 2 \times (3 \times (4 \times 1))) \\ &\Rightarrow \text{facti}(1 - 1, 1 \times (2 \times (3 \times (4 \times 1)))) \\ &\Rightarrow 1 \times (2 \times (3 \times (4 \times 1))) \\ &\vdots \\ &\Rightarrow 24 \end{aligned}$$

图2-4 惰性求值的空间漏洞

大多数惰性求值的程序设计语言都是纯函数式的。在ML中是通过命令来进行输入输出的，惰性求值能和命令组合在一起吗？它可能会在不可预知的时候对子表达式进行求值，在这种情况下几乎不可能写出可靠的程序。很多研究都指出应该将函数式程序设计和命令式程序设计结合在一起（Peyton Jones和Wadler，1993）。

书写递归函数

由于递归在函数式程序设计中是很基本的，因此让我们花些时间来剖析几个递归函数。程序设计并没有魔术般的公式可循，不过也许可以通过例子来学习。我们已经看过的这个递归函数实现了欧几里得算法（辗转相除法）：

```
fun gcd(m,n) =
  if m=0 then n
  else gcd(n mod m, m);
> val gcd = fn : int * int -> int
```

两个整数的最大公因子的定义是可以整除这两个数的最大整数。欧几里得算法是正确的，因为整除 m 和 n 的数和整除 m 和 $n-m$ 的一样，并且经过重复地相减，就和能整除 m 和 $n \bmod m$ 的一样了。我们可以看看算法的效率

$$\begin{aligned} \text{gcd}(5499, 6812) &\Rightarrow \text{gcd}(1313, 5499) \Rightarrow \text{gcd}(247, 1313) \\ &\Rightarrow \text{gcd}(78, 247) \Rightarrow \text{gcd}(13, 78) \Rightarrow \text{gcd}(0, 13) \Rightarrow 13 \end{aligned}$$

欧几里得算法要追溯到古代了。我们未必能赶上积淀了两千年的智慧，不过我们应该朝着同样典雅和高效的解法而努力。

递归涉及到将问题简化为更小的子问题。效率的关键是选择适当的子问题。这样的子问题一定不会很多，而剩下的计算应该就是很简单的了。

2.14 整数次幂

显然的计算 x^k 的方法是反复地乘以 x 。使用递归，问题 x^k 可以被简化成子问题 x^{k-1} 。但是 x^{10} 不用计算10次乘法，我们可以先计算 x^5 ，然后将其平方。由于 $x^5 = x \times x^4$ ，我们同样可以通过平方计算 x^4 ：

$$x^{10} = (x^5)^2 = (x \times x^4)^2 = (x \times (x^2)^2)^2$$

通过使用定律 $x^{2n} = (x^n)^2$ ，我们已经比重复的乘法进步很多了，然而计算过程仍旧比较混乱，使用 $x^{2n} = (x^2)^n$ 省去了嵌套的平方：

$$2^{10} = 4^5 = 4 \times 16^2 = 4 \times 256^1 = 1024$$

通过这个方法， power 对于实数 x 和大于0的整数 k 进行了 x^k 的计算：

```
fun power(x,k) : real =
  if k=1 then x
  else if k mod 2 = 0 then      power(x*x, k div 2)
  else x * power(x*x, k div 2);
> val power = fn : real * int -> real
```

注意 mod 是怎样判断指数的奇偶的。整数除法（ div ）中，当 k 是奇数时，截取商的整数部分。函数 power 体现了下面的等式（当 $n > 0$ 时）：

$$x^1 = x$$

$$x^{2n} = (x^2)^n$$

$$x^{2n+1} = x \times (x^2)^n$$

我们可以用内置的指数函数`Math.pow`来核对一下`power`的结果:

```
power(2.0, 10);
> 1024.0 : real
power(1.01, 925);
> 9937.353723 : real
Math.pow(1.01, 925.0);
> 9937.353723 : real
```

将 x^{2n} 简化成 $(x^2)^n$ 而不是 $(x^n)^2$ 使得`power`的第一个递归调用成为迭代的。第二个调用（当指数为奇数时）只能通过添加一个存储结果的参数来变成迭代的，这会增加不必要的复杂性。

练习 2.12 写出`power(2.0, 29)`的计算步骤。

练习 2.13 在最坏的情况下，`power(x, k)`要进行多少次乘法？

练习 2.14 为什么不选择 $k=0$ 作为递归的基本情形，而用 $k=1$ 呢？

2.15 斐波那契数列

斐波那契数列0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ..., 由于拥有众多奇妙的性质，对于数学爱好者来说是非常熟悉的。数列 (F_n) 定义如下

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-2} + F_{n-1} \quad \text{当 } n \geq 2 \text{ 时}$$

相应的递归函数是一个评测编译器生成代码效率的基准。对于其他的用途来说，用它就太慢了，因为这个方法不断地重复计算同样的子问题。例如，由于

$$F_8 = F_6 + F_7 = F_6 + (F_5 + F_6)$$

它把 F_6 算了两遍。

每一个斐波那契数都是前面两个的和:

$$0 + 1 = 1 \quad 1 + 1 = 2 \quad 1 + 2 = 3 \quad 2 + 3 = 5 \quad 3 + 5 = 8 \dots$$

所以我们应该用序偶来计算。函数`nextfib`以 (F_{n-1}, F_n) 为参数，返回下一个序偶 (F_n, F_{n+1}) 。

```
fun nextfib(prev, curr : int) = (curr, prev+curr);
> val nextfib = fn : int * int -> int * int
```

特殊名字`it`代表了上一个序偶，可以帮助我们来演示这个函数:

```
nextfib (0, 1);
> (1, 1) : int * int
nextfib it;
> (1, 2) : int * int
nextfib it;
> (2, 3) : int * int
nextfib it;
> (3, 5) : int * int
```


通过递归将`nextfib`应用必要的次数:

```
fun fibpair (n) =
  if n=1 then (0,1) else nextfib(fibpair(n-1));
> val fibpair = fn : int -> int * int
```

它可以非常迅速地计算出 (F_{29}, F_{30}) , 如果照原先的办法可能需要接近三百万次的函数调用:

```
fibpair 30;
> (514229, 832040) : int * int
```

我们来仔细地看看为什么`fibpair`是正确的。很显然 $\text{fibpair}(1) = (F_0, F_1)$ 。如果当 $n > 1$ 时下式成立

$$\text{fibpair}(n) = (F_{n-1}, F_n)$$

那么则有

$$\text{fibpair}(n+1) = (F_n, F_{n-1} + F_n) = (F_n, F_{n+1})$$

刚才看到了一个关于 $\text{fibpair}(n) = (F_{n-1}, F_n)$ 的证明, 使用了数学归纳法 (mathematical induction)。我们将在第6章看到更多这样的证明。证明函数式程序的性质往往是很直接的, 这也是函数式语言的一个主要优点。

函数`fibpair`使用了一个正确且相当有效的算法来计算斐波那契数, 并展示出如何使用序偶来进行计算。但是它的递归模式浪费了空间: `fibpair`构造了嵌套调用

$$\text{nextfib}(\text{nextfib}(\cdots \text{nextfib}(0, 1) \cdots))$$

为了使算法成为迭代的, 让我们把计算过程反过来, 由里到外地进行:

```
fun itfib (n, prev, curr) : int =
  if n = 1 then curr      (* 对于n = 0的情况不适用 *)
  else itfib (n-1, curr, prev+curr);
> val itfib = fn : int * int * int -> int
```

函数`fib`则使用正确的初值来调用`itfib`:

```
fun fib (n) = itfib(n, 0, 1);
> val fib = fn : int -> int
fib 30;
> 832040 : int
fib 100;
> 354224848179261915075 : int
```

对于斐波那契数来说, 迭代要比递归更清楚些:

$$\text{itfib}(7, 0, 1) \Rightarrow \text{itfib}(6, 1, 1) \Rightarrow \cdots \text{itfib}(1, 8, 13) \Rightarrow 13$$

在6.3节中, 我们将通过证明一个不太寻常的定律 $\text{itfib}(n, F_k, F_{k+1}) = F_{k+n}$ 来证明`itfib`是正确的。

练习 2.15 F_n 的递归定义中的重复计算和传名调用规则有怎样的关系? 惰性求值能有效地执行这个定义吗?

练习 2.16 证明用递归定义来运算 F_n 所需的步骤数是以 n 为指数的。`fib`的运算次数呢? 假设使用传值调用。

练习 2.17 $\text{itfib}(n, F_{k-1}, F_k)$ 的值是什么?

因子。

```
fun fraction (n,d) = (n div gcd(n,d), d div gcd(n,d));
```

对 $\text{gcd}(n, d)$ 的重复计算是浪费的, 可以通过预先定义一个辅助函数来避免:

```
fun divideboth (n, d, com: int) = (n div com, d div com);
fun fraction (n,d) = divideboth (n, d, gcd(n,d));
```

但是通过这种办法来将 $\text{gcd}(n, d)$ 命名为 com 有点绕弯了。ML允许在一个表达式里面声明几个名字:

```
fun fraction (n,d) =
  let val com = gcd(n,d)
  in (n div com, d div com) end;
> val fraction = fn : int * int -> int * int
```

上面使用了`let`表达式, 它的一般形式是

53

`let D in E end`

在求值过程中, 首先对声明 D 进行求值: 对声明中的表达式进行求值, 并且将其结果命名。这样建立起来的环境仅在`let`表达式中是可见的。然后再对表达式 E 进行求值, 它的值作为整个表达式的值返回。

通常 D 是复合声明, 也就是说包含一系列的声明:

$$D_1; D_2; \dots; D_n$$

每个声明的作用在后续的声明中都是可见的。这里分号是可选的, 很多程序员都会省略它们。

2.17 例子: 实数平方根

牛顿-拉夫森方法可以找到函数的根: 换句话说, 它可以求解形如 $f(x) = 0$ 的方程。在已知一个好的初始近似值的情况下, 它会收敛得很快。这个方法对于计算平方根非常有效, 只要解方程 $a - x^2 = 0$ 就可以了。要计算 \sqrt{a} , 就要选择任意一个正数 x_0 , 比如1, 作为第一个近似值。如果 x 是当前的近似值, 那么下一个近似值就是 $(a/x+x)/2$ 。当相邻近似值的差足够小的时候就可以停止计算了。

函数`findroot`实现了这个算法, x 是 a 的平方根的近似值, 而 acc 是相对于 x 的精度。由于相邻的近似值要用到几次, 我们使用`let`来给`nextx`命名。

```
fun findroot (a, x, acc) =
  let val nextx = (a/x + x) / 2.0
  in if abs (x-nextx) < acc*x
    then nextx else findroot (a, nextx, acc)
  end;
> val findroot = fn : (real * real * real) -> real
```

函数`sqroot`使用适当的初值来调用`findroot`。

```
fun sqroot a = findroot (a, 1.0, 1.0E~10);
> val sqroot = fn : real -> real
sqroot 2.0;
> 1.414213562 : real
it*it;
> 2.0 : real
```

嵌套的函数声明。我们的平方根函数仍不够理想。参数 a 和 acc 毫无变化地在每次递归调用 $findroot$ 的时候传递着。相对于 $findroot$ ，它们应该是全局的，这样更有效、更清楚。

增加一层 let 声明将 $findroot$ 嵌在 $sqroot$ 里面。首先声明精度 acc ，使它在 $findroot$ 中是可见的，参数 a 当然也是可见的。

54

```
fun sqroot a =
  let val acc = 1.0E~10
      fun findroot x =
          let val nextx = (a/x + x) / 2.0
          in if abs (x-nextx) < acc*x
             then nextx else findroot nextx
          end
      in findroot 1.0 end;
  > val sqroot = fn : real -> real
```

如同我们从ML的回应所知，在 $sqroot$ 外 $findroot$ 是不可见的。

大多数声明都可以在 let 里面使用。值、函数、类型和异常都是可以声明的。

什么时候不用 let 。考虑要取得 $f(x)$ 和 $g(x)$ 的最小值。当然可以将这些量用 let 来命名：

```
let val a = f x
    val b = g x
in
  if a < b then a else b
end
```

更好的做法是声明一个函数来取得两个实数中最小的那个：

```
fun min(a,b) : real = if a < b then a else b;
```

现在 $min(f\ x, g\ x)$ 的意思非常清楚了，因为 min 计算了一个人们非常熟悉的东西。抓住每个机会来定义有意义的函数，哪怕只用它们一次。

2.18 使用 $local$ 来隐藏声明

$local$ 声明很像 let 表达式：

```
local  $D_1$  in  $D_2$  end
```

这个声明就像是系列声明 $D_1; D_2$ 一样，只不过 D_1 只在 D_2 内可见，在外面不可见。由于一系列的声明会被看作是一个声明，因此 D_1 和 D_2 都可以声明多个名字。

let 被经常地使用，而 $local$ 则很少用到。 $local$ 的唯一用途就是隐藏声明。回想计算斐波那契数的 $itfib$ 和 fib 。函数 $itfib$ 只应该被 fib 调用：

55

```
local
  fun itfib (n, prev, curr) : int =
    if n=1 then curr
    else itfib (n-1, curr, prev+curr)
in
  fun fib (n) = itfib(n,0,1)
end;
> val fib = fn : int -> int
```

在这里， $local$ 声明将 $itfib$ 变为 fib 私有的了。

练习 2.20 上面我们使用 $local$ 隐藏了 $itfib$ 。为什么不简单地将 $itfib$ 嵌在 fib 之内呢？对比一下

上面对`findroot`和`sqroot`的处理。

练习 2.21 使用`let`，我们可以在整数平方根函数中省掉费时的平方操作。编写一个`introot`的变体，由 n 得到它的整数平方根 k ，并和差 $n - k^2$ 配对作为序偶结果。这只需要简单的乘法和除法，一个优化的编译器可以将它们转换为位操作。

2.19 联立声明

一个联立声明同时定义多个名字。通常声明是彼此独立的。但是`fun`声明允许递归，因此联立声明可以引入相互递归的函数。

如下形式的`val`声明

```
val Id1 = E1 and ... and Idn = En
```

首先对表达式 E_1, \dots, E_n 求值，然后声明标识符 Id_1, \dots, Id_n ，让它们具有相应的值。由于声明在所有表达式求值结束之前还不起作用，所以它们的顺序是无关紧要的。

下面我们声明了 π 、 e 和2的自然对数。

```
val pi    = 4.0 * Math.atan 1.0
and e     = Math.exp 1.0
and log2  = Math.ln 2.0;
> pi = 3.141592654 : real
> e  = 2.718281828 : real
> log2 = 0.693147806 : real
```

只用一个输入就定义了三个名字。这个联立声明强调了它们是相互独立的。

56

让我们定义大笨钟的钟鸣声：

```
val one = "BONG ";
> val one = "BONG " : string
val three = one^one^one;
> val three = "BONG BONG BONG " : string
val five = three^one^one;
> val five = "BONG BONG BONG BONG BONG " : string
```

这里我们必须依次进行这三个声明。

联立声明也可以交换名字的值：

```
val one = three and three = one;
> val one = "BONG BONG BONG " : string
> val three = "BONG " : string
```

当然，这么干有点不太正常，不过它却体现出声明是同时进行的。如果是顺序进行的话就会赋予`one`和`three`相同的绑定。

相互递归的函数。有些函数是相互递归的 (mutually recursive)，它们是相互基于对方来递归定义的。递归下降语法分析器就是个典型的例子。这种语法分析器对于语法里面的每一个元素都定义一个函数，而大多数的语法就是相互递归的：ML的声明可以包含表达式，同时表达式也可以包含声明。遍历由此生成的语法分析树的函数也将是相互递归的。

语法分析和树都会在本书后面的章节中讲述。这里举一个简单的例子，看一下级数求和

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots + \frac{1}{4k+1} - \frac{1}{4k+3} \dots$$

通过相互递归，和的最后一项可以是正的也可以是负的：

```
fun pos d = neg(d-2.0) + 1.0/d
and neg d = if d>0.0 then pos(d-2.0) - 1.0/d
              else 0.0;
> val pos = fn : real -> real
> val neg = fn : real -> real
```

这一下声明了两个函数。我们看到级数慢慢地收敛了：

```
4.0 * pos(201.0);
> 3.151493401 : real
4.0 * neg(8003.0);
> 3.141342779 : real
```

相互递归的函数通常都可以借助加入一个参数来合并成一个函数：

57

```
fun sum (d, one) =
    if d > 0.0 then sum(d-2.0, ~one) + one/d else 0.0;
> val sum = fn : real * real -> real
```

这样， $sum(d, 1.0)$ 返回和 $pos(d)$ 同样的结果，而 $sum(d, -1.0)$ 则返回和 $neg(d)$ 同样的结果。

仿真goto语句。函数式程序设计语言和过程式程序设计语言要比你想像的更为相像。任何赋值语句和goto语句的组合——最糟糕的过程式代码——都可以被翻译成一组相互递归的函数。下面是一个简单的例子：

```
var x := 0; y := 0; z := 0;
F: x := x+1; goto G
G: if y<z then goto F else (y := x+y; goto H)
H: if z>0 then (z := z-x; goto F) else stop
```

为每一个标号， F 、 G 和 H ，声明相互递归的函数。每一个函数的参数就是一个代表其全部变量的元组。

```
fun F(x,y,z) = G(x+1,y,z)
and G(x,y,z) = if y<z then F(x,y,z) else H(x,x+y,z)
and H(x,y,z) = if z>0 then F(x,y,z-x) else (x,y,z);
> val F = fn : int * int * int -> int * int * int
> val G = fn : int * int * int -> int * int * int
> val H = fn : int * int * int -> int * int * int
```

运行前，调用 $F(0, 0, 0)$ 赋给 x 、 y 和 z 相应的初值，并返回相当于过程式代码所得到的结果。

```
F(0,0,0);
> (1, 1, 0) : int * int * int
```

函数式的程序是引用透明的，不过也可以是完全不透明的。如果你的代码变得和上面相似的话，就要小心了！

练习 2.22 下面的声明有什么作用？

```
val (pi, log2) = (log2, pi);
```

练习 2.23 考虑在 $n > 1$ 上定义的数列 (P_n)

$$P_n = 1 + \sum_{k=1}^{n-1} P_k$$

58

(特别地, $P_1 = 1$ 。)用ML函数表达这个计算。它的效率怎样? 有没有更快的方法来计算 P_n 呢?

模块系统初步

工程师是基于设备的部件来了解一个设备的, 类似地, 可以通过基于部件的子部件来了解部件。自行车有轮子; 轮子有轴; 而轴有轴承, 依此类推。需要经过几个阶段我们才能看到单独的金属和塑料的那个层次。通过这种方式, 就可以在一个抽象的层次上来了解整个自行车, 也可以详细地了解它的部件。工程师一般都是通过改装其中的某个部件来改善设计, 而不用同时考虑其他部件。

程序(要比自行车复杂得多!)也应该被看作是由部件组成的。传统上, 子程序就是一个过程或者函数, 但是这些组件都太小了, 就像是把自行车看成是由千百块各种各样的金属组成的一样。许多近期的语言把程序看作是由模块(module)组成的, 每个模块都定义了自己的数据结构和相关操作。每个模块的接口也都是与模块本身分开进行描述的。因此, 不同的模块可以由项目组的不同成员编写, 而编译器可以检测每个模块是否符合它的接口描述。

看看我们关于向量的例子。函数`addvec`单独提出来没什么用处, 它必须和其他的向量操作一起使用, 这些操作共享了向量的表达方式。我们可以猜测其他操作是相关的, 因为它们的名字都以`vec`结尾的, 但是并没有什么来强制这种命名协定。它们应该组合在一起形成一个程序模块。

ML的结构(structure)将相关的类型、值以及其他的结构组合在一起, 遵守一个统一的命名准则。ML的签名(signature)则通过列出每个组件中的名字和类型(或者其他的属性)描述了一类结构。

在其他的语言中, 也有类似Standard ML中的签名和结构的概念, 比如Modula-2中的定义和实现模块(Wirth, 1985)。ML也提供函子(functor)——以其他结构作为参数的结构——不过我们将这些推到第7章再讲。

2.20 复数

很多数学对象是可以进行加减乘除运算的。除了熟悉的整数和实数以外, 还有有理数、矩阵、多项式等。我们下面的例子将是复数, 它在科学数学中是很重要的。我们将复数的算术运算通过结构`Complex`凑在一起, 然后为`Complex`声明一个签名, 这个签名也符合任何其他声明了同样算术运算的结构。这将提供一个泛型算术运算的基础。

我们先来简单地介绍一下复数。一个复数(complex number)形如 $x + iy$, 其中, x 和 y 是实数, 而 i 是一个规定满足 $i^2 = -1$ 的常数。这样, x 和 y 就确定了一个复数。

复数零是 $0 + i0$ 。两个复数的和由相应的 x 和 y 的和组成; 它们的差也是类似的。积和倒数的定义看上去复杂些, 但通过代数定律以及公理 $i^2 = -1$ 也不难确定:

$$(x + iy) + (x' + iy') = (x + x') + i(y + y')$$

$$(x + iy) - (x' + iy') = (x - x') + i(y - y')$$

$$(x + iy) \times (x' + iy') = (xx' - yy') + i(xy' + x'y)$$

$$1/(x + iy) = (x - iy) / (x^2 + y^2)$$

在上面的倒数中, y 部分是 $-y/(x^2 + y^2)$ 。我们现在可以将复数的商 z/z' 定义为 $z \times (1/z')$ 。

像定义向量那样, 我们可以通过下面的定义来实现复数

```
type complex = real*real;
val complexzero = (0.0, 0.0);
:
```

不过, 最好还是使用结构。

❶ 进一步的阅读。Penrose (1989) 更为详尽地解释了复数系统, 给出了大量的动机和例子。他讨论了复数和分形之间的联系, 包括了一个Mandelbrot集合的定义。在书的后面, 复数在其量子力学的讨论中扮演了主要的角色。Penrose赋予了复数一种形而上学的意义, 不过这个也许不可全信! Feynman等 (1963) 在第22章给出了一个更技术性的, 但是非常出色且赏心悦目的关于复数的描述。

2.21 结构

声明可以被包含在关键字`struct`和`end`之间来组合在一起形成一个结构。这个组合的结果可以通过`structure`结构声明绑定到一个标识符上:

60

```
structure Complex =
  struct
    type t = real*real;
    val zero = (0.0, 0.0);
    fun sum ((x,y), (x',y')) = (x+x', y+y') : t;
    fun diff ((x,y), (x',y')) = (x-x', y-y') : t;
    fun prod ((x,y), (x',y')) = (x*x' - y*y', x*y' + x'*y) : t;
    fun recip (x,y) =
      let val t = x*x + y*y
      in (x/t, ~y/t) end
    fun quo (z,z') = prod(z, recip z');
  end;
```

当结构`Complex`可见时, 其组件可以通过复合名字来访问, 例如`Complex.zero`和`Complex.sum`。在结构体内, 组件通过原有的标识符来访问, 例如`zero`和`sum`, 可以留意一下在声明`quo`的时候所使用的`recip`。复数的类型是`Complex.t`。如果声明结构的目的在于定义一个类型, 那么这个类型通常都命名为`t`。

我们可以放心地使用短名字。它们不会和其他结构里面的名字冲突。标准库非常依赖于这一点, 比如区分绝对值函数`Int.abs`和`Real.abs`。

让我们试验一下新的结构。声明两个ML标识符, `i`和`a`; 数学家一般会将它们相应地写作 i 和 0.3 。

```
val i = (0.0, 1.0);
> val i = (0.0, 1.0) : real * real
val a = (0.3, 0.0);
> val a = (0.3, 0.0) : real * real
```

分两步计算 $a + i + 0.7$, 结果应该等于 $1 + i$ 。最后, 将这个数平方得到 $2i$:

```
val b = Complex.sum(a,i);
> val b = (0.3, 1.0) : Complex.t
```



```
Complex.sum(b, (0.7, 0.0));
> (1.0, 1.0) : Complex.t
Complex.prod(it,it);
> (0.0, 2.0) : Complex.t
```

我们看到`Complex.t`和`real × real`的类型是一样的, 更为混乱的是, 它和之前的`vec`类型也是一样的。第7章描述了如何定义抽象类型 (abstract type), 其内部表示是隐藏的。

61 结构看来和记录有点像, 但它们有很大的差别。记录的组成部分只能是值 (也许包含其他记录)。而结构的组成部分则可以包括类型和异常 (当然也可以是其他的结构)。然而, 你却不能用结构来进行计算, 因为只有在程序模块进行连接的时候才会创建结构。结构应该被看作是一种封装了的环境。

2.22 签名

签名是结构里面每一个组件的描述。在对结构`Complex`作出声明之后, ML通过打印出它所导出的相应的签名来作为回应:

```
structure Complex = ...;
> structure Complex :
>   sig
>     type t
>     val diff : (real * real) * (real * real) -> t
>     val prod : (real * real) * (real * real) -> t
>     val quo : (real * real) * (real * real) -> t
>     val recip : real * real -> real * real
>     val sum : (real * real) * (real * real) -> t
>     val zero : real * real
>   end
```

关键字`sig`和`end`括住了签名体。它显示了所有值的类型, 也提到了类型`t`。(有些编译器显示`eqtype t`, 而不是`type t`, 来提示`t`是一个所谓的相等类型。)

由ML编译器导出的签名往往不是最适合我们使用的。结构里面可能包含一些应该是私有的定义。通过声明我们自己的签名, 同时去掉那些私有的名字, 就可以将它们隐藏起来, 而不被结构的使用者看到。比如, 这里我们可能需要隐藏`recip`。

上面打印出来的签名有时将复数类型表示为`t`, 有时又表示为`real × real`。如果到处都使用`t`, 我们将得到一个通用的签名, 它描述了类型`t`, 以及相关的运算`sum`、`prod`等:

```
signature ARITH =
  sig
    type t
    val zero : t
    val sum : t * t -> t
    val diff : t * t -> t
    val prod : t * t -> t
    val quo : t * t -> t
  end;
```

62 这个声明将名字`ARITH`赋给了包括在`sig`和`end`之间的签名。我们还可以定义其他的结构并使它们遵守签名`ARITH`。这里是一个有理数结构的框架:

```
structure Rational : ARITH =
  struct
```

```

type t   = int*int;
val zero = (0, 1);
:
end;

```

签名描述了那些ML想安全地将程序单元连接起来所需的信息。它不能描述组件实际上是做什么的。具有出色文档的签名包含了描述每个组件目的的注释。而描述组件实现的注释应该放在结构中，而不是签名里。签名可以通过多种方式组合成新的签名，结构也可以有类似的组合。

ML的函子可以表示泛型模块：例如，一个能接受任何遵守签名`ARITH`的结构的模块。标准库为这方面提供了广泛的可能性。ML系统可以提供多种精度的浮点数，只要结构与签名`FLOAT`匹配。数值算法可以编码成函子。将该函子应用到某个精度的浮点数结构上，就能构造出该精度上的数值算法实例。由此来说，ML拥有面向对象语言，比如C++的一些功能——不过是以一种更严格的形式，因为结构并不是可计算的值。[⊖]

练习 2.24 声明一个结构`Real`，遵守签名`ARITH`，也就是说，`Real.t`就是类型`real`，且组件`zero`、`sum`、`prod`等代表了相应的实数运算。

练习 2.25 完成上面的有理数结构声明，此声明要基于定律 $n/d + n'/d' = (nd' + n'd)/dd'$ ， $(n/d) \times (n'/d') = nn'/dd'$ 以及 $1/(n/d) = d/n$ 。利用`gcd`函数来保持最简分式，并保证分母永远是正的。

多态类型检测

直到最近，对于类型检测的争论一直在僵持着，主要有两个强硬的立场：

- 弱类型语言像Lisp和Prolog给予程序员在书写大型程序时所需要的自由。
- 强类型语言像Pascal通过限制程序员的自由使他们少犯错误以提供安全性。

63

多态类型检测提供了一个新的立场：既有强类型的安全性，同时也非常灵活方便。程序里不再堆满了类型描述，因为大多数类型信息都是自动推导出来的。

类型表示了值的集合。函数的参数类型规定了哪些值是可以作为参数的。其结果类型则说明了哪些值是可作为结果被函数返回的。因此，`div`若用一对整数作为参数的话，其返回结果也只能是整数。如果除数为0的话则根本没有结果：取代结果的是一个错误信息。就算是在这种意外情况下，函数`div`也是忠实于它的类型的。

ML也可以赋予恒等函数（identity function）以类型，使这个函数直接返回它的参数。因为恒等函数可以应用在任何类型的参数上，所以它是多态的（polymorphic）。总的来说，如果一个对象具有多种类型，那么它就是多态的。ML的多态性是基于类型模式（type scheme）的，这类似于类型的模式或模板（template）。例如，恒等函数具有类型模式： $\alpha \rightarrow \alpha$ 。

2.23 类型推导

在没有或几乎没有显式类型信息的情况下，ML可以推导出函数声明涉及到的所有类型。类型推导是顺着自然而严格的过程进行的。ML标记出所有常量的类型，并且将类型检测规则应用到每种形式的表达式上。在整个声明中，不同地方出现的同一变量要具有相同的类型。

⊖ C++的程序员可能已经注意到了，这种泛型程序设计的设施类似于C++中的模板。不过，作者在写这本书的时候C++大概还没有像现在这么成熟。——译者注

重载的操作符（像+）的类型则必须根据上下文来确定。

下面是条件表达式的类型检测规则。如果 E 具有 $bool$ 类型，且 E_1 和 E_2 具有相同的类型，比如 τ ，那么

```
if E then E1 else E2
```

也具有类型 τ 。否则，该表达式就会出现类型错误。

让我们逐步地分析 $facti$ 的类型检测：

```
fun facti (n,p) =
  if n=0 then p else facti(n-1, n*p);
```

常量0和1具有 int 类型。由于 $n = 0$ 和 $n - 1$ 都涉及到了整数，因此 n 也具有 int 类型。现在 $n*p$ 必定是整数乘法了，所以 p 也具有 int 类型。因为 p 是 $facti$ 的返回结果，所以函数的结果类型是 int ，而参数类型是 $int \times int$ 。这个分析方法也适用于递归调用。经过所有的这些检测，ML可以回应

```
> val facti = fn : int * int -> int
```

如果这些类型不一致的话，编译器就拒绝这个声明。

练习 2.26 叙述 $itfib$ 的类型检测步骤。

练习 2.27 检测下面函数声明的类型：

```
fun f (k,m) = if k=0 then 1 else f(k-1);
```

2.24 多态函数声明

如果类型推导还剩下一些无约束的类型，那么声明就是多态的，通俗地说，“具有多种形态”。大多数多态函数都涉及序偶、表和其他数据结构。它们通常都是做一些简单的事情，例如将一个值和它自己配对：

```
fun pairself x = (x,x);
> val pairself = fn : 'a -> 'a * 'a
```

这个类型是多态的，因为里面包含了类型变量（type variable），命名为 $'a$ 。在ML里，类型变量以撇号（单引号）开始。

```
'b      'c      'we_band_of_brothers      '3
```

让我们用 α 、 β 、 γ 来代表ML的类型变量 $'a$ 、 $'b$ 、 $'c$ ，因为传统上类型变量是用希腊字母表示的。书写 $x : \tau$ 表示“ x 具有 τ 类型”，例如， $pairself : \alpha \rightarrow (\alpha \times \alpha)$ 。顺便提一下， \times 比 \rightarrow 的优先级高； $pairself$ 的类型可以写作 $\alpha \rightarrow \alpha \times \alpha$

多态类型是一个类型模式。用类型去替换类型变量形成了一个类型模式的实例（instance）。一个具有多态类型的值可以具有无限多的类型。当 $pairself$ 应用到实数上时，它实际是具有类型 $real \rightarrow real \times real$ 的。

```
pairself 4.0;
> (4.0, 4.0) : real * real
```

应用到一个整数上， $pairself$ 就相当于具有类型 $int \rightarrow int \times int$ 。

```
pairself 7;
> (7, 7) : int * int
```

下面`pairself`应用到一个序偶上, 结果称为`pp`。

```
val pp = pairself ("Help!", 999);
> val pp = (("Help!", 999), ("Help!", 999))
> : (string * int) * (string * int)
```

投影函数返回序偶的一个分量。函数`fst`返回第一个分量, 函数`snd`返回第二个:

```
fun fst (x,y) = x;
> val fst = fn : 'a * 'b -> 'a
fun snd (x,y) = y;
> val snd = fn : 'a * 'b -> 'b
```

在考虑它们的多态类型之前, 我们先把它们应用到`pp`上:

```
fst pp;
> ("Help!", 999) : string * int
snd (fst pp);
> 999 : int
```

`fst`的类型是 $\alpha \times \beta \rightarrow \alpha$, 里面有两个类型变量。参数序偶可以涉及任意两个类型 τ_1 和 τ_2 (不一定要不同), 而结果只具有 τ_1 类型。

多态函数可以用来表示其他函数。从 $((x, y), w)$ 返回 x 的函数可以直接编写, 也可以连续应用两次`fst`:

```
fun fstfst z = fst (fst z);
> val fstfst = fn : ('a * 'b) * 'c -> 'a
fstfst pp;
> "Help!" : string
```

我们应该知道`fstfst`的类型是 $(\alpha \times \beta) \times \gamma \rightarrow \alpha$ 。注意一点, 多态函数在同一个表达式的不同地方可以有不同的类型。上面的内层`fst`具有类型 $(\alpha \times \beta) \times \gamma \rightarrow (\alpha \times \beta)$, 而外层的`fst`则具有类型 $\alpha \times \beta \rightarrow \alpha$ 。

现在来点奇怪的看看: 下面的函数在做什么?

```
fun silly x = fstfst (pairself (pairself x));
> val silly = fn : 'a -> 'a
```

其实也没干什么:

```
silly "Hold off your hands.";
> "Hold off your hands." : string
```

它的类型, $\alpha \rightarrow \alpha$, 暗示了`silly`是一个恒等函数。这个函数可以更为直接地表达为:

```
fun I x = x;
> val I = fn : 'a -> 'a
```

❶ 进一步的阅读。Milner (1978) 给出了一个多态类型检测的算法, 并且证明了一个类型正确的程序不会遭受运行时类型错误的困扰。Damas和Milner (1982)^①还证明了这个算法推导出的类型是基本的 (principal): 类型是尽可能多态的。Cardelli和Wegner (1985) 较全面地评述了多态性的几种实现方法, 对于Standard ML来说是相当复杂的。

① 根据作者的勘误表: 这篇文章仅给出了证明的框架, 证明本身在Damas (1985) 的博士论文中。
——译者注

相等测试在有限的程度上是多态的：它为大多数（但不是全部）的类型进行了定义。Standard ML提供了一类相等类型变量（equality type variable）来概括这些被限制的类型。见3.14节。

回想一下某些内置的函数是重载的：比如同时为整数和实数定义了加法（+）。重载和多态性相处不易。它将类型检测的算法复杂化了，并且经常要求程序员书写类型约束。幸亏只有几个重载函数。而程序员是不能引入更多重载的。

要点小结

- 变量代表某个值，它可被再声明，但不能被更新。
- 基本的值具有类型`int`、`real`、`char`、`string`或`bool`。
- 任何类型的值都可以组合成元组和记录。
- 数值运算可以被表示成递归函数。
- 迭代函数以有限的方式使用递归，其中递归调用基本上只是跳转。
- 结构和签名可以为组织大的程序服务。
- 多态类型是包含类型变量的类型模式。

第3章 表

在一次公开演讲中，C. A. R. Hoare (1989a) 描述了他的一个算法，用来在一堆整数中寻找第*i*小的整数。这个算法很精巧，不过Hoare通过将它比喻成一个纸牌游戏而把算法叙述得非常清楚，令人叹服。每一张纸牌代表一个整数，将纸牌根据规则从一堆移到另一堆，可以很快找到满足要求的那个整数。

然后Hoare改变了游戏的规则。每一张纸牌占据一个固定的位置，并且仅有的移动是和另一张纸牌交换位置。这就是在叙述基于数组的算法了。数组具有很高的效率，但也是有代价的。数组可能会难倒大部分的听众，如同它会难倒富有经验的程序员一样。Mills和Linger (1986) 声称如果数组仅限于栈、队列等没有下标寻址的操作的话，程序员可以变得更有成效。

函数式程序员通常用表来处理数据的聚合。就像Hoare的那叠纸牌，表只允许每次处理一个数据项，这是非常清晰的。表是很容易从数学的角度去理解的，并且实际上比通常想像中的要高效。

本章提要

本章叙述了怎样在Standard ML中进行关于表的程序设计。它提供了几个通常涉及到使用数组的例子，例如矩阵操作和排序。

本章包括了以下几节：

- 表的简介。介绍了表的记法。Standard ML是通过模式匹配来对表进行操作的。
- 基本的表函数。介绍了一系列的函数。这些都是对表的程序设计具有指导性的例子，也是解决更难的问题时所不可缺少的。
- 表的应用。一些逐渐深入的例子讲述了可以利用表来解决的多种问题。
- 多态函数中的相等测试。通过例子介绍和演示了相等的多态性。其中包括了一组实用的有限集合上的函数。
- 排序：案例分析。过程式程序设计和函数式程序设计在效率上的比较。在一个试验中，过程式程序仅比清晰得多的函数式程序快一点儿。
- 多项式算术。计算机可以解代数问题。以符号的形式利用表来进行多项式的加法、乘法和除法。

69

表的简介

表 (list) 是一个有限的元素序列。典型的表如`[3, 5, 9]`和`["fair", "Ophelia"]`。空表，`[]`，里面没有任何元素。元素的顺序是有意义的，并且元素可以重复出现。例如，下面的表是互不相同的：

`[3, 4]` `[4, 3]` `[3, 4, 3]` `[3, 3, 4]`

表中的元素可以是任意类型的，包括元组甚至其他的表。一个表中的每个元素必须具有相同的类型。假设元素的类型是 τ ，那么表的类型就是 τ list。如此

```
[(1, "One"), (2, "Two"), (3, "Three")] : (int * string) list
[ [3.1], [], [5.7, ~0.6] ]           : (real list) list
```

空表，`[]`，具有多态类型 α list。可以把它看作具有任意类型的元素。

注意，类型操作符`list`使用后缀语法。它比 \times 和 \rightarrow 的优先级更高。因此，类型 $\text{int} \times \text{string list}$ 相当于 $\text{int} \times (\text{string list})$ ，而不是 $(\text{int} \times \text{string}) \text{ list}$ 。另外 int list list 相当于类型 $(\text{int list}) \text{ list}$ 。

3.1 表的构造

每一个表都是仅由两个原语构造的：常量`nil`以及中缀操作符`::`，读作“cons”，意思是构造（construct）。

- `nil`是空表，`[]`，的同义词。

- 操作符`::`通过在已有的表前加入一个元素来构造一个新的表。

每个表要么是空表`nil`，要么形如 $x :: l$ ，其中 x 是表头（head）， l 是表尾（tail）。表尾本身也是一个表。表的操作是不对称的：表的首元素要比最后的元素容易到达得多。

70

如果 l 是表 $[x_1, \dots, x_n]$ ，并且 x 是具有正确类型的值，那么， $x :: l$ 则是表 $[x, x_1, \dots, x_n]$ 。构造新的表并不会影响 l 的值。表`[3, 5, 9]`是如下这样构造的：

```
nil = []
9 :: [] = [9]
5 :: [9] = [5, 9]
3 :: [5, 9] = [3, 5, 9]
```

可以看到，元素是从反方向放进去的。表`[3, 5, 9]`可以写成多种形式，例如 $3 :: (5 :: (9 :: \text{nil}))$ 或 $3 :: (5 :: [9])$ 或 $3 :: [5, 9]$ 。为了避免总是写括号，中缀操作符“cons”被定义为右结合的。记法 $[x_1, x_2, \dots, x_n]$ 代表了 $x_1 :: x_2 :: \dots :: x_n :: \text{nil}$ 。元素也可以由表达式给出，下面是一个包含多个实数表达式的表

```
[ Math.sin 0.5, Math.cos 0.5, Math.exp 0.5 ];
> [0.479425539, 0.877582562, 1.64872127] : real list
```

表的记法可以构造包含固定数目元素的表。看一下怎样构造一个从 m 到 n 的整数的表：

```
[m, m + 1, ..., n]
```

首先要比较 m 和 n ，如果 m 比 n 大，那么在 m 和 n 之间就没有数，表将是个空表。否则，表头元素是 m ，而表尾则是表 $[m + 1, \dots, n]$ 。递归地构造了表尾之后，结果就可以由下式得出

```
m :: [m + 1, ..., n]
```

这个过程对应于下面这个简单的ML函数：

```
fun upto (m,n) =
  if m>n then [] else m :: upto(m+1,n);
> val upto = fn : int * int -> int list
  upto(2,5);
> [2, 3, 4, 5] : int list
```

i 其他语言中的表。弱类型语言像Lisp和Prolog使用序偶来表示表，比如 $(3, (5, (9, "nil")))$ 。这里"nil"是一种结束标记，而所表达的是表 $[3, 5, 9]$ 。这种表达方式在ML里是不行的，因为这样的话，表的类型就会随着元素个数的变化而变化。那样的话，*upto*是什么类型呢？

ML里的表的语法和Prolog里的有微妙的差别。在Prolog里面，表 $[5 | [6]]$ 和表 $[5, 6]$ 是一样的。而在ML里， $[5 :: [6]]$ 则是表 $[[5, 6]]$ 。

71

3.2 表的操作

表（像元组一样）是一种结构值。在ML里，元组上的函数可以将它的参数写成一个模式，以此显示出参数的结构，以及命名结构的分量。作用在表上的函数也可以类似地书写。例如，

```
fun prodof3 [i,j,k] : int = i*j*k;
```

声明了一个将表里的数相乘的函数，但是只适用于恰好有三个数的表！

表的操作通常是递归定义的，分几种情形来处理。怎样求得表内所有数的乘积呢？

- 如果表是空的，那么积就是1（根据约定）。
- 如果表不空，那么积就是表头乘以表尾的积。

这个在ML中可以这样表示：

```
fun prod [] = 1
  | prod (n::ns) = n * (prod ns);
> val prod = fn : int list -> int
```

函数声明分成两句，用竖线（|）分开。每一个分句处理一个模式。可能会有几个分句和复杂的模式，前提是类型要相同。由于模式涉及到表，而结果又有一种可能是整数1，因此ML推导出*prod*是将一个整数表映射到一个整数的函数。

```
prod[2,3,5];
> 30 : int
```

最常见的表的分情是将其分为空表和非空表。不过寻找一个表中整数的最大值这样的问题就有点不同了，因为空表是没有最大值的。对于这个问题可以分为两种情况

- 单元素表 $[m]$ 的最大值是 m 。
- 要寻找两个或以上元素的表 $[m, n, \dots]$ 的最大值，只要从表中删除 m 和 n 中较小的那个元素，然后递归地寻找剩下元素组成的表的最大值。

根据这个算法得到了ML函数

```
fun maxl [m] : int = m
  | maxl (m::n::ns) = if m>n then maxl(m::ns)
                      else maxl(n::ns);
> ***Warning: Matches are not exhaustive
> val maxl = fn : int list -> int
```

72

注意这个警告信息：ML检测到了*maxl*对于参数为空表的情形是没有定义的。另外也可以看到模式 $m :: n :: ns$ 是如何描述形如 $[m, n, \dots]$ 的表的。较小的元素在递归调用时被去掉了。

这个函数对于空表以外的参数都起作用。


```
maxl [ ~4, 0, ~12];
> 0 : int
maxl [];
> Exception: Match
```

异常 (exception), 眼下可以被看作是个运行错误。异常是因为函数 *maxl* 被应用到了一个它没有定义的参数上。通常, 异常都会终止程序运行, 不过它们也可以被捕获, 关于这方面的内容将在下一章介绍。

中间 (临时) 表。有时在计算过程中会生成及使用一些表。例如, 阶乘函数可以利用 *prod* 和 *upto* 来定义:

```
fun factl (n) = prod (upto (1,n));
> val factl = fn : int -> int
factl 7;
> 5040 : int
```

这个声明简练而清晰, 避免了显式的递归。构造表 $[1, 2, \dots, n]$ 的开销也许并不是最重要的。重要的是, 函数式程序设计应该促进对于程序正确性的证明, 而这一点在这里没体现出来。对于阶乘的基本定律

$$\text{factl}(m+1) = (m+1) \times \text{factl}(m)$$

并没有一个明显的证明。展开函数的定义, 我们有

$$\text{factl}(m+1) = \text{prod}(\text{upto}(1, m+1)) = ?$$

下一步并不清楚, 因为回顾 *upto* 的定义, 里面的递归是跟随第一个而不是第二个参数的。那个老老实实的递归的阶乘函数定义看来更加清楚。

字符串和表。表在字符串处理上非常重要。大多数的函数式语言提供字符类型, 然后将字符串看作是字符的表。在新的标准库中, ML 引入了字符类型, 但是并没有把字符串看作是表。内置函数 *explode* 将字符串转换成字符的表, 而函数 *implode* 则进行相反的操作, 将表中的字符按顺序连接成字符串。

73

```
explode "Banquo";
> [#"B", #"a", #"n", #"q", #"u", #"o"] : char list
implode it;
> "Banquo" : string
```

类似地, 函数 *concat* 将一个字符串表中的所有成员连成一个字符串。

基本的表函数

给定一个表, 可以取得它的长度、提取出第 n 个元素、取得前缀 (前一部分) 或后缀 (后一部分), 以及翻转元素的顺序。给定两个表, 可以将一个追加到另一个后面, 或者, 如果它们长度一样, 可以将相应的元素配对。在这一节中声明的函数都是不可缺少的, 并将在本书后面大量使用。所有这些函数都是多态的。

这里效率将作为重点考虑。对于某些函数, 直接的递归定义不如迭代的版本效率高, 但对于另一些函数, 迭代的方式既削弱了可读性也降低了效率。

3.3 表的测试和分解

三个基本的表函数是`null`、`hd`和`tl`。

`null`函数。这个函数测试一个表是不是空表：

```
fun null [] = true
  | null (_::_) = false;
> val null = fn : 'a list -> bool
```

函数是多态的：测试表是否为空并不涉及到分析表的元素。第二个模式中的下划线(`_`)是一个占位符，表示该位置的值在语句中是用不到的。这些下划线称为通配符(wildcard)，可以使我们不必为用不到的部分起名字。

`hd`函数。函数返回非空表的表头元素：

```
fun hd (x::_) = x;
> ***Warning: Patterns not exhaustive
> val hd = fn : 'a list -> 'a
```

这个模式使用了通配符代表表尾，而表头命名为`x`。由于没有代表空表的模式，因此ML打印出了警告信息。像`max1`一样，这是一个部分函数。

下面我们有一个表的表，它的表头本身是一个表，表头的表头是一个整数。每次使用`hd`都会去掉一层方括号。

```
hd[[[1,2], [3]], [[4]]];
> [[1, 2], [3]] : (int list) list
hd it;
> [1, 2] : int list
hd it;
> 1 : int
```

如果再来一次`hd it`；会怎样？

`tl`函数。函数返回非空表的表尾。记住，表尾是包含了除去表头以外的所有表元素的子表。

```
fun tl (_::xs) = xs;
> ***Warning: Patterns not exhaustive
> val tl = fn : 'a list -> 'a list
```

像`hd`一样，这也是个部分函数，它总是返回另一个表：

```
tl ["Out", "damned", "spot!"];
> ["damned", "spot!"] : string list
tl it;
> ["spot!"] : string list
tl it;
> [] : string list
tl it;
> Exception: Match
```

通过`null`、`hd`和`tl`，所有其他的表函数都可以不使用模式匹配来书写。计算整数表内所有元素的积的函数可以这样写：

```
fun prod ns = if null ns then 1
               else (hd ns) * (prod (tl ns));
```

如果你喜欢这个版本的`prod`,你可能会放弃ML而选择Lisp。为了增加清晰性,Lisp的原语里面有CAR和CDR。一般的人都会认为模式匹配比`hd`和`tl`更清楚。好的ML编译器可以通过分析一系列的`模式`来给函数生成最好的代码。更重要的是,编译器可以在`模式`没有涵盖函数中所有可能的参数时给出警告信息。

练习 3.1 不用模式匹配,而用`null`、`hd`和`tl`书写函数`maxl`。

75 练习 3.2 书写一个返回表的最后一个元素的函数。

3.4 与数量有关的表处理

现在来声明函数`length`、`take`和`drop`。它们的功能如下:

$$l = [\underbrace{x_0, \dots, x_{i-1}}_{take(i)}, \underbrace{x_i, \dots, x_{n-1}}_{drop(i)}] \quad length(l) = n$$

`length`函数。表的长度可以直接用递归来求得:

```
fun nlength [] = 0
  | nlength (x::xs) = 1 + nlength xs;
> val nlength = fn : 'a list -> int
```

它的类型 $\alpha \text{ list} \rightarrow \text{int}$,表明了`nlength`可以应用到包含任何类型元素的表上。我们把它用在一个表的表上:

```
nlength[[1,2,3], [4,5,6]];
> 2 : int
```

你会觉得答案应该是6吗?

虽然`nlength`是正确的[⊖],不过把它应用到很长的表上面时,其浪费的空间也是不堪忍受的:

$$\begin{aligned} nlength[1, 2, 3, \dots, 10000] &\Rightarrow 1 + nlength[2, 3, \dots, 10000] \\ &\Rightarrow 1 + (1 + nlength[3, \dots, 10000]) \\ &\vdots \\ &\Rightarrow 1 + (1 + 9998) \\ &\Rightarrow 1 + 9999 \Rightarrow 10000 \end{aligned}$$

那些“一”堆积了起来,浪费和表长度成正比的空间,并且可能导致程序异常终止运行。迭代版本的函数要好得多,它利用另一个参数来进行累加:

```
local
  fun addlen (n, []) = n
    | addlen (n, x::l) = addlen (n+1, l)
in
  fun length l = addlen (0, l)
end;
> val length = fn : 'a list -> int
length (explode "Throw physic to the dogs!");
> 25 : int
```

函数`addlen`将表的长度加到另一个初值为0的数上。由于它没有别的用途,所以将其作为`length`的局部声明。它的执行过程是:

[⊖] `nlength`的“n”在这里代表naïve,暗示这是一个朴素而幼稚的实现。——译者注

$$\begin{aligned}
 \text{addlen}(0, [1, 2, 3, \dots, 10000]) &\Rightarrow \text{addlen}(1, [2, 3, \dots, 10000]) \\
 &\Rightarrow \text{addlen}(2, [3, \dots, 10000]) \\
 &\vdots \\
 &\Rightarrow \text{addlen}(10000, []) \Rightarrow 10000
 \end{aligned}$$

对于效率的显著改善补偿了可读性上的不足。

take 函数。调用 *take(l, i)* 返回由表 *l* 的前 *i* 个元素组成的表：

```

fun take ([], i)      = []
  | take (x::xs, i) = if i>0 then x::take(xs, i-1)
                      else [];
> val take = fn : 'a list * int -> 'a list
take (explode "Throw physic to the dogs!", 5);
> ["T", "h", "r", "o", "w"] : char list

```

下面是一个计算过程的例子：

$$\begin{aligned}
 \text{take}([9, 8, 7, 6], 3) &\Rightarrow 9 :: \text{take}([8, 7, 6], 2) \\
 &\Rightarrow 9 :: (8 :: \text{take}([7, 6], 1)) \\
 &\Rightarrow 9 :: (8 :: (7 :: \text{take}([6], 0))) \\
 &\Rightarrow 9 :: (8 :: (7 :: [])) \\
 &\Rightarrow 9 :: (8 :: [7]) \\
 &\Rightarrow 9 :: [8, 7] \\
 &\Rightarrow [9, 8, 7]
 \end{aligned}$$

我们看到 $9 :: (8 :: (7 :: []))$ 是一个表达式而不是一个值。对这个表达式求值的结果是构造了表 $[9, 8, 7]$ 。分配必需的空间是要花时间的，特别是还要考虑到后来所引至的垃圾收集的时间。确实，*take* 花费了大部分的时间用来构造它的结果。

很遗憾，对于 *take* 的递归调用是越来越深的，就像刚才的 *nlength* 一样。让我们再尝试写一个迭代的版本，把结果累积在另一个参数里面：

```

fun rtake ([], _, taken) = taken
  | rtake (x::xs, i, taken) =
      if i>0 then rtake(xs, i-1, x::taken)
      else taken;
> val rtake = fn : 'a list * int * 'a list -> 'a list

```

这个函数递归得很浅也很漂亮……

$$\begin{aligned}
 \text{rtake}([9, 8, 7, 6], 3, []) &\Rightarrow \text{rtake}([8, 7, 6], 2, [9]) \\
 &\Rightarrow \text{rtake}([7, 6], 1, [8, 9]) \\
 &\Rightarrow \text{rtake}([6], 0, [7, 8, 9]) \\
 &\Rightarrow [7, 8, 9]
 \end{aligned}$$

但是，结果却是倒的！

如果倒过来的结果可以接受的话，函数 *rtake* 还是值得考虑的。不过，即便是在 *take* 中，递归的大小同输出的大小相比也是可以忍受的。这里要考虑的是，*nlength* 返回的是一个整数，而 *take* 返回的是一个表。构造一个表是很慢的，这可能比由于深度递归而临时消耗的空间更为

重要。讲求效率的目的是要将开销减少到合乎比例。

`drop`函数。`drop(l, i)`返回的表包含了表`l`中除去前`i`个元素以外的其余元素:

```
fun drop ([], _) = []
  | drop (x::xs, i) = if i>0 then drop (xs, i-1)
                      else x::xs;
> val drop = fn : 'a list * int -> 'a list
```

很幸运, 这里的递归调用显然是迭代的。

```
take (["Never", "shall", "sun", "that", "morrow", "see!"], 3);
> ["Never", "shall", "sun"] : string list
drop (["Never", "shall", "sun", "that", "morrow", "see!"], 3);
> ["that", "morrow", "see!"] : string list
```

练习 3.3 `take(l, i)`和`drop(l, i)`在 $i > \text{length}(l)$ 且 $i < 0$ 时都返回什么? (库函数版本将产生异常。)

练习 3.4 写一个`nth(l, n)`函数, 返回`l`中的第`n`个元素 (从0开始计数)。

3.5 追加和翻转

中缀操作符`@`将一个表追加到另一个表后面; 函数`rev`将一个表翻转。它们都是内置的函数, 不过它们的定义却值得进行详细地分析。

追加操作。追加是将一个表的元素加到另一个表的后面:

78

$$[x_1, \dots, x_m] @ [y_1, \dots, y_n] = [x_1, \dots, x_m, y_1, \dots, y_n]$$

什么样的递归可以完成这个呢? 传统上`append`这个名字表示了操作是在一个表的后面发生的, 但是我们的表却总是要从前面来构造。下面定义的基本思想要上溯到Lisp的早期:

```
infixr 5 @;
fun ([] @ ys) = ys
  | ((x::xs) @ ys) = x :: (xs@ys);
> val @ = fn : 'a list * 'a list -> 'a list
```

函数的类型是 $\alpha \text{ list} \times \alpha \text{ list} \rightarrow \alpha \text{ list}$, 表明函数接受两个元素类型相同的表作为参数, 比如, 两个字符串的表, 或是两个 (同类型的) 表的表:

```
["Why", "sinks"] @ ["that", "cauldron?"];
> ["Why", "sinks", "that", "cauldron?"] : string list
[[2,4,6,8], [3,9]] @ [[5], [7]];
> [[2, 4, 6, 8], [3, 9], [5], [7]] : int list list
```

`[2, 4, 6] @ [8, 10]`的计算过程是如下进行的:

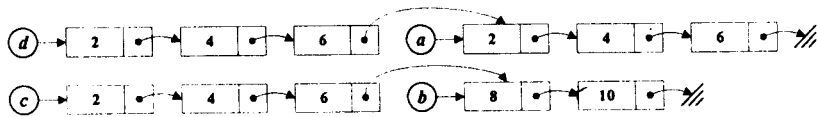
$$\begin{aligned} [2, 4, 6] @ [8, 10] &\Rightarrow 2 :: ([4, 6] @ [8, 10]) \\ &\Rightarrow 2 :: (4 :: ([6] @ [8, 10])) \\ &\Rightarrow 2 :: (4 :: (6 :: ([] @ [8, 10]))) \\ &\Rightarrow 2 :: (4 :: (6 :: [8, 10])) \\ &\Rightarrow 2 :: (4 :: [6, 8, 10]) \\ &\Rightarrow 2 :: [4, 6, 8, 10] \\ &\Rightarrow [2, 4, 6, 8, 10] \end{aligned}$$

后面的那三步把第一个表的各元素放在了第二个表前面。和*take*的情形一样，构造新表的代价超过了递归深度所付出的代价；因此迭代的版本就不必要了。计算 $xs @ ys$ 的开销和 xs 的长度成正比，而且完全与 ys 无关。甚至是 $xs @ []$ 也会将 xs 复制一遍。

在Pascal和C中，可以利用指针类型来实现表类型，并且通过改变其中一个表的尾指针，让它指向另外一个表，来将两个表连接起来。破坏性的更新比复制要快很多，但是如果不小心的话，你的表就会有麻烦了。如果两个表碰巧是同一个指针来表示的话将会怎样？ML的表含有使用安全的内部指针。如果你喜欢冒险，ML也有显式的指针类型，详见第8章。

ML的表的内部表示是程序员很熟悉的单向链接表，我们很容易怀疑为什么连接两个表的开销和第二个表无关？既然要复制，为什么不复制第二个表呢？

假设 $a = [2, 4, 6]$ ， $b = [8, 10]$ ， $c = a @ b$ ， $d = a @ a$ ：



我们看到， a 的元素被复制到了 b 和 a 的前面分别形成了 c 和 d ，当然 a 没受到任何破坏，同时 b 、 c 和 d 也是好的表， a 、 b 、 c 和 d 所代表的值都存在。在理解函数式语言的时候要注意把重点放在值上面，而不是内存单元上面。例如，尽管 b 和 c 使用了共同的内存单元，但它们很成功地表达了两个值。一个值不同于内存单元，它的总体和局部都是不会被更新的，所以共用部分没有问题。利用更新尾指针来实现连接两个表的Pascal和C的版本虽然快，却使 a 原来的值从内存中消失了（更不用说这个方法根本不能构造出 d ）。——译者注

*rev*函数。表的翻转可以利用追加操作来定义。把表头元素变成翻转后的表尾：

```
fun nrev []      = []
  | nrev (x::xs) = (nrev xs) @ [x];
> val nrev = fn : 'a list -> 'a list
```

这样做的效率显然是很低的。如果 $nrev$ 的参数是一个长度为 n ($n > 0$)的表，追加操作需要调用 $cons(·)$ 正好 $n-1$ 次来复制翻转后的表尾。构造表 $[x]$ 也需要一次 $cons$ ，总共需要 n 次调用。而递归翻转表尾又需要 $n-1$ 次的 $cons$ 调用，依此类推，最终需要的 $cons$ 调用次数为：

$$0+1+2+\cdots+n = \frac{n(n+1)}{2}$$

代价是二次的：和 n^2 成正比。

我们在*rtake*里面已经看到了另一种翻转表的方法：不断地将一个表的元素移向另一个表。

```
fun revAppend ([], ys) = ys
  | revAppend (x::xs, ys) = revAppend (xs, x::ys);
> val revAppend = fn : 'a list * 'a list -> 'a list
```

这里没有用到追加操作。翻转的步骤数和要翻转的表的长度是成正比的。这个函数类似追加函数，只不过是把第一个参数给翻转了。

```
revAppend (["Macbeth", "and", "Banquo"], ["all", "hail!"]);
> ["Banquo", "and", "Macbeth", "all", "hail!"] : string list
```

较为高效的翻转函数调用了`revAppend`，并传入一个空表作为第二个参数：

```
fun rev xs = revAppend(xs, []);
> val rev = fn : 'a list -> 'a list
```

这里，较长的定义换来了突出的效果。翻转一个1000个元素的表，`rev`只要调用1000次`::`，而`nrev`需要500 500次。此外，`revAppend`里的递归是迭代的。这里的关键思想——将一个表中的元素积累到另一个参数上，而不使用追加——适用于很多其他的函数。

练习 3.5 修改追加函数，使得它能高效地处理`xs @ []`的情形。

练习 3.6 如果我们在`nrev`的定义中将`[x]`改成`x`会怎样？

80 **练习 3.7** 分别给出使用`nrev`和`rev`来翻转表`[1, 2, 3, 4]`的计算过程。

3.6 表的表，序偶的表

模式匹配和多态性可以很好地处理数据结构的组合。观察一下下面这些函数的类型。

`concat`函数。这个函数将一个表内的所有元素连成一个表：

```
fun concat [] = []
  | concat (l::ls) = l @ concat ls;
> val concat = fn : 'a list list -> 'a list
concat [[ "When", "shall" ], [ "we", "three" ], [ "meet", "again" ]];
> [ "When", "shall", "we", "three", "meet", "again" ]
> : string list
```

`l @ concat ls`中的复制过程应该是很快的，因为`l`通常都比`concat ls`要短得多。

`zip`函数。这个函数将两个表中的对应元素配对：

$$\text{zip}([x_1, \dots, x_n], [y_1, \dots, y_n]) = [(x_1, y_1), \dots, (x_n, y_n)]$$

如果两个表的元素个数不等，`zip`将忽略多余的元素。这个函数的声明用到了比较复杂的模式：

```
fun zip (x::xs, y::ys) = (x, y) :: zip (xs, ys)
  | zip _ = [];
> val zip = fn : 'a list * 'b list -> ('a*'b) list
```

`zip`定义中的第二个模式使用了通配符，它能匹配所有可能的模式。但是这个通配符只有在第一个模式匹配失败以后才会被考虑。ML根据给定的顺序尝试模式匹配。

`unzip`函数。这是`zip`函数的逆函数，它把序偶的表转换成表的序偶：

$$\text{unzip}([(x_1, y_1), \dots, (x_n, y_n)]) = ([x_1, \dots, x_n], [y_1, \dots, y_n])$$

在函数式语言里同时构造两个表可能需要一些技巧。一种方法是利用辅助函数：

```
fun conspair ((x, y), (xs, ys)) = (x::xs, y::ys);

fun unzip [] = ([], [])
  | unzip (pair::pairs) = conspair(pair, unzip pairs);
```

81 在一个`let`声明中利用模式匹配来将递归调用的结果分解开来，就可以省去这个`conspair`了：

```
fun unzip [] = ([], [])
  | unzip ((x, y)::pairs) =
    let val (xs, ys) = unzip pairs
    in (x::xs, y::ys) end;
> val unzip = fn : ('a*'b) list -> 'a list * 'b list
```

一个迭代的函数可以同时为它的参数构造多个结果。这是`unzip`一个表的最简单的方法，不过返回的表却是翻转的。

```
fun rev_unzip([], xs, ys)      = (xs, ys)
  | rev_unzip((x,y)::pairs, xs, ys) =
    rev_unzip(pairs, x::xs, y::ys);
```

❶ 表和标准库。标准库提供了上面提到的大多数函数。追加（中缀`@`）、翻转（`rev`）、`null`、`hd`、`tl`和`length`都可以在顶层环境中使用。`List`结构还提供了`take`、`drop`和`concat`，以及其他函数。`ListPair`结构则提供了`zip`和`unzip`。

请尽量使用标准库提供的这些函数。我们自己定义的函数缺少错误处理。库函数会在遇到错误输入的情况下抛出异常，例如`List.Empty`（如果你试图提取空表的表头元素）和`Subscript`（如果你试图`take`比表的长度更多的元素）。库函数也会为了效率而进行优化。

练习 3.8 将下面的函数与`concat`进行比较，看看它的作用和效率：

```
fun f []      = []
  | f(l::ls)  = f(ls)
  | f((x::l)::ls) = x :: f(l::ls);
```

练习 3.9 给出`zip`函数的等价定义，要求和模式匹配的顺序无关。

练习 3.10 `rev(rtake(l, i, []))`会比`take(l, i)`效率更高吗？请考虑所有的开销。

表的应用

这一节演示了怎样利用表去完成相对复杂的任务，例如二进制算术和矩阵操作。如果你愿意的话，尽管跳过那些较难的例子。

这里也解决了来自《A Discipline of Programming》(Dijkstra, 1976) 这一经典著作中的两个问题。Dijkstra展示了程序“令人叹服的、深刻理性的美”。他的程序使用的是数组，表会不会更具美感呢？

82

3.7 找零钱

让我们从简单的开始：找零钱。这个任务是要将一定的钱数用一些硬币来表达，这些硬币可能具有的价值是通过一个表来给定的。自然地，我们希望所找的零钱使用尽可能大的硬币。如果硬币的价值是按降序给出的话，这就很容易：

```
fun change (coinvals, 0)      = []
  | change (c::coinvals, amount) =
    if amount < c then change(coinvals, amount)
    else c :: change(c::coinvals, amount-c);
> ***Warning: Patterns not exhaustive
> val change = fn : int list * int -> int list
```

这个定义几乎是最直接的了。如果目标钱数是零，则不需要硬币；如果最大的硬币`c`太大，则放弃它；否则就使用这个硬币，并将钱数减去`c`后继续找零钱。

让我们声明两个表来表达英美两国的硬币价值:

```
val gb_coins = [50,20,10,5,2,1]
and us_coins = [25,10,5,1];
```

这样一来, 43便士和43美分的表达方式就不同了:

```
change(gb_coins, 43);
> [20, 20, 2, 1] : int list
change(us_coins, 43);
> [25, 10, 5, 1, 1, 1] : int list
```

不过找零钱并没有一开始看上去那么简单。假如我们只有价值为5和2的硬币, 那怎么办呢?

```
change([5,2], 16);
> Exception: Match
```

编译器在我们声明`change`的时候已经警告过有这种可能性了。但是16是很容易用5和2的组合来表示的! 我们的算法是贪婪式的: 它总是首选价值最大的硬币, 试图将16表达成 $5 + 5 + 5 + c$, 这样只剩下 $c = 1$ 这样一种不可能的情形了。

能不能设计个好一点的算法呢? 回溯 (backtracking) 的意思是当错误发生时倒退最近的一步, 并重新另试。一种实现回溯的方法是利用异常, 这会在4.8节中讲到。另一种方法是计算出所有可能答案的表。观察下面是怎样使用`coins`来存放到目前为止所选取的硬币的。

83

```
fun allChange (coins, coinvals, 0)          = [coins]
  | allChange (coins, [], amount)          = []
  | allChange (coins, c::coinvals, amount) =
    if amount < 0 then []
    else allChange(c::coins, c::coinvals, amount-c) @
          allChange(coins, coinvals, amount);
> val allChange = fn
> : int list * int list * int -> int list list
```

那句“patterns not exhaustive”的警告不见了; 说明函数考虑了所有的情况。如果给定一个无解的问题, 函数则会返回一个空表, 而不是抛出异常:

```
allChange([], [10,2], 27);
> [] : int list list
```

我们来试几个更有意思的例子:

```
allChange([], [5,2], 16);
> [[2, 2, 2, 5, 5], [2, 2, 2, 2, 2, 2, 2, 2]]
> : int list list
allChange([], gb_coins, 16);
> [[1, 5, 10], [2, 2, 2, 10], [1, 1, 2, 2, 10], ...]
> : int list list
```

一共有25种办法来找16便士的零钱! 乍一看, 这种办法是站不住脚的。为了控制指数般增长的解的数目, 可以使用惰性求值, 只产生必要的解 (见5.14节)。

❶ 进一步的阅读。一般来讲, 找零钱问题要比显见的困难得多。它与子集和 (subset-sum) 问题关系密切, 这是个NP完全问题。这表示了很可能找不到一个有效的算法来确定一个钱数能否用一套给定的硬币来表达。Cormen等 (1990) 讨论了求得近似解的算法。

练习 3.11 书写一个函数来将整数转化成罗马数字表示。输入合适的参数，你的函数应该可以将1984表达成MDCCCCLXXXIII或者MCMLXXXIV。

练习 3.12 找零钱函数期望`coinvals`包含严格递减的正整数值。如果这个前提条件不满足将会怎样？

练习 3.13 我们很少会幸运得有无数的硬币可用。修改`allChange`来从一个有限的钱包中找零钱。

练习 3.14 修改`allChange`，使用额外的参数来积累它的解，从而省掉那个追加操作。通过找99便士的零钱来比较一下它和原来版本的效率。

84

3.8 二进制算术

函数式程序设计可能看上去远离硬件，不过表可以很好地仿真数字电路。下面定义了元素为0和1的表的加法和乘法。

加法。如果你忘却了二进制加法的规则，请看看下面二进制版本的 $11 + 30 = 41$ ：

$$\begin{array}{r} 11110 \\ + 1011 \\ \hline 101001 \end{array}$$

加法从右到左进行。两个位再加上（从右边的）任何进位得出一个该位置的和位和一个向左的进位。从右到左对于表来说方向就不对了，表头元素在最左边。所以二进制位将按逆序存储。

两个二进制数的长度可能有所不同。如果其中一个二进制位表结束了，那么它留下的进位就要传递给另一个表剩下的那些位。

```
fun bincarry (0, ps)      = ps
  | bincarry (1, [])      = [1]
  | bincarry (1, p::ps) = (1-p) :: bincarry(p, ps);
> ***Warning: Patterns not exhaustive
> val bincarry = fn : int * int list -> int list
```

没错，模式可以包含常量：整数、实数、布尔量和字符串。函数`bincarry`可以传播进位0或1，也只可能是这两个进位值。对于其他的进位值，函数是无定义的。

二进制和是定义在两个位表和一个进位上的。当其中一个位表结束时，就用`bincarry`去处理另外那个。如果有两个位需要相加，那么就计算它们的和及其进位：

```
fun binsum (c, [], qs)      = bincarry (c, qs)
  | binsum (c, ps, [])      = bincarry (c, ps)
  | binsum (c, p::ps, q::qs) =
    ((c+p+q) mod 2) :: binsum((c+p+q) div 2, ps, qs);
> val binsum = fn
> : int * int list * int list -> int list
```

让我们试试 $11 + 30 = 41$ ，记住二进制位是按逆序排列的：

```
binsum(0, [1,1,0,1], [0,1,1,1,1]);
> [1, 0, 0, 1, 0, 1] : int list
```

乘法。二进制乘法是通过移位和加法完成的。例如， $11 \times 30 = 330$ ：

85

$$\begin{array}{r}
 11110 \\
 \times \quad 1011 \\
 \hline
 11110 \\
 11110 \\
 + \quad 11110 \\
 \hline
 101001010
 \end{array}$$

可以通过插入一个0来完成移位:

```

fun binprod ([], _) = []
| binprod (0::ps, qs) = 0::binprod(ps, qs)
| binprod (1::ps, qs) = binsum(0, qs, 0::binprod(ps, qs));
> ***Warning: Patterns not exhaustive
> val binprod = fn : int list * int list -> int list

```

我们来计算 $11 \times 30 = 330$:

```

binprod([1,1,0,1], [0,1,1,1,1]);
> [0, 1, 0, 1, 0, 0, 1, 0, 1] : int list

```

二进制算术的结构。在大的程序中，通过命名规则，例如前缀`bin`，来将相关的函数联系起来的方式并不怎么样。二进制算术的函数应该组合在一起形成一个结构，比如说`Bin`。在结构里面可以使用短一些的名字，使得代码更为易读。从外部看，结构的组件拥有统一的复合名字。上面的函数声明可以很容易地封装成一个结构:

```

structure Bin =
struct
  fun carry (0, ps) = ...
  fun sum (c, [], qs) = ...
  fun prod ([], _) = ...
end;

```

再花一点力气的话，结构`Bin`就可以做成满足2.22节的签名`ARITH`。这将使得二进制数的运算符与复数的运算符具有完全相同的接口。如此一来，就可以把二进制算术收集到那些可以用于泛型算术运算包的算术结构中去了。但是，二进制算术和复数算术很不一样；例如，除法不是精确的。需要注意的是，我们有责任搞清楚在某一种特定的情况下都需要哪些性质。

86

练习 3.15 书写一些函数，通过布尔量的表来完成二进制的加法和乘法，不能使用内置的算术函数。

练习 3.16 书写一个函数来完成两个二进制数的除法。

练习 3.17 利用上一个练习的结果，或者书写哑函数来扩展结构`Bin`，使其满足签名`ARITH`。

练习 3.18 十进制数可以用0到9组成的整数表来表示。书写一些函数来完成二进制数和十进制数（表）的相互转换。并计算100的阶乘。

3.9 矩阵的转置

矩阵可以看成是由行组成的表，而每一行又是由矩阵的一行元素组成的表。矩阵

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$$

可以在ML中声明成:

```
val matrix = [ ["a","b","c"],
                ["d","e","f"] ];
> val matrix = [["a", "b", "c"], ["d", "e", "f"]]
> : string list list
```

使用这种表示方法, 矩阵的转置可以做得很好, 因为它是顺着行和列完成的, 中间没有跳跃。转置函数将行组成的表

$$A = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ \vdots & \vdots & & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix}$$

变换成列组成的表:

$$A^T = \begin{bmatrix} x_{11} & \cdots & x_{n1} \\ x_{12} & \cdots & x_{n2} \\ \vdots & & \vdots \\ x_{1m} & \cdots & x_{nm} \end{bmatrix}$$

一种转置的办法就是不断地从矩阵里面提取列。每行的头一个元素合在一起组成了矩阵的第一列:

87

```
fun headcol [] = []
  | headcol ((x::_) :: rows) = x :: headcol rows;
> ***Warning: Patterns not exhaustive
> val headcol = fn : 'a list list -> 'a list
```

同样, 每行的表尾合在一起组成了矩阵剩下的那些列:

```
fun tailcols [] = []
  | tailcols ((_::xs) :: rows) = xs :: tailcols rows;
> ***Warning: Patterns not exhaustive
> val tailcols = fn : 'a list list -> 'a list list
```

再看一下用在这个小矩阵上的效果:

```
headcol matrix;
> ["a", "d"] : string list
tailcols matrix;
> [["b", "c"], ["e", "f"]] : string list list
```

调用headcol和tailcols将矩阵砍成下面的样子:

$$\left(\begin{array}{c|cc} a & b & c \\ d & e & f \end{array} \right)$$

这两个函数导致了一个不寻常的递归: tailcols以n个表组成的表作为参数, 返回n个短一点的表所组成的表。这个过程最后将得到n个空表, 并在此时结束。

```
fun transp ([]::rows) = []
  | transp rows = headcol rows :: transp (tailcols rows);
> val transp = fn : 'a list list -> 'a list list
transp matrix;
> [["a", "d"], ["b", "e"], ["c", "f"]] : string list list
```

转置后的矩阵是:

$$\begin{pmatrix} a & d \\ b & e \\ c & f \end{pmatrix}$$

❶ 更加精炼的方法。这里展示的很多程序都可以通过使用高阶函数更为简练地表达, 例如 *map*, 它将一个函数应用到表中的每一个元素上。我们稍后就会讨论到高阶函数。你可以瞥一下5.7节, 那里重新考虑了矩阵的转置。

练习 3.19 *headcol*和*tailcols*不能处理什么样的输入模式? 如果“矩阵”的行不一样长, *transp*将返回什么呢?

88 **练习 3.20** 输入一个空表, *transp*将怎么办? 请解释一下。

练习 3.21 另外写一个转置函数, 使用将行转为列来取代将列转为行的方法。

3.10 矩阵乘法

我们快速地复习一下矩阵乘法。两个向量的点积 (dot product) (或称为内积) 是这样的

$$(a_1, \dots, a_k) \cdot (b_1, \dots, b_k) = a_1 b_1 + \dots + a_k b_k$$

如果 A 是 $m \times k$ 的矩阵, 而 B 是 $k \times n$ 的矩阵, 那么它们的积 (product) $A \times B$ 就是一个 $m \times n$ 的矩阵。对于每一个 i 和 j , $A \times B$ 的第 (i, j) 个元素就是 A 的第 i 行与 B 的第 j 列的点积。例如:

$$\begin{pmatrix} 2 & 0 \\ 3 & -1 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 2 \\ 4 & -1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 4 \\ -1 & 1 & 6 \\ 4 & -1 & 0 \\ 5 & -1 & 2 \end{pmatrix}$$

上面的元素 $(1, 1)$ 是由下式计算出来的

$$(2, 0) \cdot (1, 4) = 2 \times 1 + 0 \times 4 = 2$$

在下面的点积函数中, 两个向量必须具有相同的长度; 当有些情况没有被涵盖时, ML 就会打印出警告信息。以后这些警告信息通常都会被略去。

```
fun dotprod([], []) = 0.0
  | dotprod(x::xs, y::ys) = x*y + dotprod(xs, ys);
> ***Warning: Patterns not exhaustive
> val dotprod = fn : real list * real list -> real
```

如果 A 只有一行, 那么 $A \times B$ 也将只有一行。函数 *rowprod* 计算了一行与矩阵 B 的积。矩阵 B 必须以它的转置形式给出: 也就是由列组成的表, 而不是由行组成的表。

```
fun rowprod(row, []) = []
  | rowprod(row, col::cols) =
    dotprod(row, col) :: rowprod(row, cols);
> val rowprod = fn
> : real list * real list list -> real list
```

结果矩阵 $A \times B$ 的每一行都是由矩阵 A 中相应的一行与整个矩阵 B 的乘积而得来的:

```
fun rowlistprod([], cols)      = []
  | rowlistprod(row::rows, cols) =
      rowprod(row, cols) :: rowlistprod(rows, cols);
> val rowlistprod = fn
> : real list list * real list list -> real list list
```

89

矩阵的乘法函数利用 *transp* 来构造由矩阵 B 的列所组成的表:

```
fun matprod(rowsA, rowsB) = rowlistprod(rowsA, transp rowsB);
> val matprod = fn
> : real list list * real list list -> real list list
```

下面是刚才矩阵例子的声明, 省去了 ML 的回应:

```
val rowsA = [ [2.0, 0.0],
               [3.0, ~1.0],
               [0.0, 1.0],
               [1.0, 1.0] ]
and rowsB = [ [1.0, 0.0, 2.0],
               [4.0, ~1.0, 0.0] ];
```

这是它们的积:

```
matprod(rowsA, rowsB);
> [[2.0, 0.0, 4.0],
>  [~1.0, 1.0, 6.0],
>  [4.0, ~1.0, 0.0],
>  [5.0, ~1.0, 2.0]] : real list list
```

练习 3.22 矩阵的负矩阵是通过把它的每一个分量求负而得到的; 就是说

$$-\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} -a & -b \\ -c & -d \end{pmatrix}$$

书写一个求负矩阵的函数。

练习 3.23 维数相同的两个矩阵可以通过把它们的相应分量相加而完成矩阵的相加; 就是说

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} + \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} = \begin{pmatrix} a+a' & b+b' \\ c+c' & d+d' \end{pmatrix}$$

书写一个将两个矩阵相加的函数。

3.11 高斯消元法

高斯消元法是一个经典的矩阵算法, 它可能看上去并不适合函数式程序设计。这个算法 (Sedgewick, 1988) 可以计算矩阵的行列式或逆矩阵, 也可以解像下面这样的线性无关的线性联立方程组:

$$\begin{aligned} x + 2y + 7z &= 7 \\ -4w + 3y - 5z &= -2 \\ 4w - x - 2y - 3z &= 9 \\ -2w + x + 2y + 8z &= 2 \end{aligned} \quad (*)$$

90 高斯消元法依次单独处理每一个未知数。方程(*)经过适当地缩放,然后加到另一个方程上去,就可以把未知数 w 从那个方程中消去。重复这个被称为主元消去(pivoting)的操作,最后联立方程组就被简化成了三角形:

$$\begin{aligned} -4w + 3y - 5z &= -2 \\ x + 2y + 7z &= 7 \\ 3y - z &= 14 \\ 3z &= 3 \end{aligned}$$

现在解就出来了,从 $z=1$ 开始。

通过消去 w 来解方程(*)是一个好的选择,因为其系数的绝对值(4)是最大的。缩放要将方程除以这个值;而一个小的除数(更不用说零!)也可能导致数值误差。函数`pivotrow`从一个由行组成的表中返回行首元素绝对值最大的一行。

```
fun pivotrow [row] = row : real list
  | pivotrow (row1::row2::rows) =
    if abs(hd row1) >= abs(hd row2)
    then pivotrow(row1::rows)
    else pivotrow(row2::rows);
> val pivotrow = fn : real list list -> real list
```

如果被选择的行有首元素 p ,那么`delrow(p , rows)`会将这行从行表中删去。[⊖]

```
fun delrow (p, []) = []
  | delrow (p, row::rows) = if Real==(p, hd row) then rows
    else row :: delrow(p, rows);
> val delrow = fn : 'a * 'a list list -> 'a list list
```

函数`scalarprod`(数量积)将一行或者说一个向量乘以一个常数 k :

```
fun scalarprod (k, []) = [] : real list
  | scalarprod (k, x::xs) = k*x :: scalarprod(k, xs);
> val scalarprod = fn : real * real list -> real list
```

函数`vectorsum`将两行或两个向量相加:

```
fun vectorsum ([], []) = [] : real list
  | vectorsum (x::xs, y::ys) = x+y :: vectorsum(xs, ys);
> val vectorsum = fn : real list * real list -> real list
```

在`gausselim`内部声明的函数`elimcol`通过行首 p (第一个系数)和行尾 $prow$ 来引用主元行。给定一个行表, `elimcol`将其中的每一行加上恰当缩放后的 $prow$ 。每行相加后的第一个元素是零,但是这些零是不用计算的;第一列只是简单地消失了。

91

```
fun gausselim [row] = [row]
  | gausselim rows =
    let val p::prow = pivotrow rows
    fun elimcol [] = []
```

⊖ 注意此处的类型变量 $'a$,它要求类型是相等类型(稍后会介绍)。最新的Standard ML基本库把实数从相等类型中除去了(可能是出于实数不能精确表示的考虑),使得等号不能用于实数的比较。不过基本库的`Real`结构中提供了实数专用的比较函数(==和!=等)。因此还是可以定义`delrow`的,不过这样定义的函数就不再是多态的了。——译者注

```

| elimcol ((x::xs)::rows) =
    vectorsum(xs, scalarprod(~x/p, prow))
    :: elimcol rows
(* 这里的rows是在elimcol定义的作用域中,
   隐藏了上面gausselim的rows *)
in (p::prow) :: gausselim(elimcol(delrow(p, rows)))
end;
> val gausselim = fn : real list list -> real list list

```

函数`gausselim`将主元行移走，消去一列，并递归地调用自己来处理简化后的矩阵。它将返回由一系列主元行组成的表，每行的长度递减，形成了一个上三角矩阵。

n 元联立方程组可以通过在 $n \times (n + 1)$ 矩阵上使用高斯消元法来求解，多出的一列是方程组的右式。解可以递归地从三角矩阵中得出。已经得出的那些解需要乘上它们的系数并加起来，这是两个向量的点积运算，然后除以该行首未知数的系数。我们不要忘了方程的右边，为了减去这个值，我们使用一个窍门：假造一个解-1，并从它开始。

```

fun solutions [] = [~1.0]
| solutions((x::xs)::rows) =
    let val solns = solutions rows
    in ~(dotprod(solns,xs)/x) :: solns end;
> val solutions = fn : real list list -> real list

```

一种理解这个定义的方法就是拿它试试上面的方程例子。我们先来计算三角矩阵：

```

gausselim [[ 0.0, 1.0, 2.0, 7.0, 7.0],
           [~4.0, 0.0, 3.0, ~5.0, ~2.0],
           [ 4.0, ~1.0, ~2.0, ~3.0, 9.0],
           [~2.0, 1.0, 2.0, 8.0, 2.0]];
> [[~4.0, 0.0, 3.0, ~5.0, ~2.0],
> [ 1.0, 2.0, 7.0, 7.0],
> [ 3.0, ~1.0, 14.0],
> [ 3.0, 3.0]] : real list list


```

忽略掉末尾假造的-1，方程的解是 $w = 3, x = -10, y = 5$ 和 $z = 1$ 。

```

solutions it;
> [3.0, ~10.0, 5.0, 1.0, ~1.0] : real list%

```

 进一步的阅读。威尔士大学的研究员们曾将Haskell语言应用于解决计算流体力学的问题。目的是为了考察一下函数式程序设计的实际用途。其中一篇文章比较了矩阵的各种表达方式(Grant等, 1996)。另一篇考虑了使用并行的可能性, 通过一个仿真的并行处理器(Grant等, 1995)。同传统的Fortran实现相比, Haskell的实现非常慢, 而且需要更多的空间; 作者们列举了一些可能增加效率的开发成果。

练习 3.24 证明如果输入的方程组是线性无关的，那么除以零的情况在`gausselim`里是不可能发生的。

练习 3.25 如果多行的行首元素绝对值都相同的话，`pivotrow`和`delrow`能正确工作吗？

练习 3.26 书写一个函数来计算矩阵的行列式。

练习 3.27 书写一个函数来求逆矩阵。

练习 3.28 书写一个满足签名 *ARITH* 的结构 *Matrix*。你可以使用上一个练习以及 3.10 节的结果，或者书写哑函数。这样矩阵也成了我们的一个算术结构了。^①

3.12 分解一个数为两个平方数之和

Dijkstra (1976) 介绍了一个程序，已知一个整数 r ，找到所有的满足方程 $x^2 + y^2 = r$ 的整数解。（假设 $x > y > 0$ ，以避免对称的解。）例如， $25 = 4^2 + 3^2 = 5^2 + 0^2$ ，而 48 612 265 则有 32 个解。

穷尽地搜索所有的 (x, y) 序偶对于大数来说并不现实，但所幸地是解有着某种规律：若 $x^2 + y^2 = r = u^2 + v^2$ 且 $x > u$ ，则必有 $y < v$ 。如果 x 从 \sqrt{r} 开始向下搜寻，同时 y 从 0 开始向上搜寻，那么在一遍扫描中就可以发现所有的解。

设 $Bet(x, y)$ 代表所有的在 x 和 y 之间的解：

$$Bet(x, y) = \{(u, v) \mid u^2 + v^2 = r \wedge x \geq u > v > y\}$$

根据下面的四条分析结果搜索合适的 x 和 y ：

1. 若 $x^2 + y^2 < r$ ，则 $Bet(x, y) = Bet(x, y + 1)$ 。当 $u \leq x$ 时，肯定不存在形如 (u, y) 的解。

2. 若 $x^2 + y^2 = r$ ，则 $Bet(x, y) = \{(x, y)\} \cup Bet(x-1, y+1)$ 。就这一个解！对于同样的 x 或 y 来说不会有其他的解了。

93

3. 若 $x^2 + y^2 > r > x^2 + (y-1)^2$ ，则 $Bet(x, y) = Bet(x-1, y)$ 。不可能有形如 (x, v) 的解了。

4. 最后，当 $x < y$ 时， $Bet(x, y) = \emptyset$ 。

这些提供了一个递归的，实际上是迭代的，搜索方法。对于第 3 种情况，如果想高效使用的话则需要特别小心。一开始，要保证 $x^2 + y^2 < r$ 成立。逐渐增加 y ，直到 $x^2 + y^2 \geq r$ 。如果 $x^2 + y^2 > r$ ，那么 y 必定是满足这个不等式最小的值，这就可以应用第 3 种情况了。将 x 减 1 就重新建立了 $x^2 + y^2 < r$ 。

一开始 $y = 0$ 且 $x = \sqrt{r}$ (r 的整数平方根)，因此开始条件是满足的。由于 $x > y$ ，我们知道在轮到减 x 之前， y 需要递增几次。因此，为了进一步提高效率，程序在内层递归之外计算 x^2 ：

```
fun squares r =
  let fun between (x,y) = (* x和y之间的所有序偶 *)
        let val diff = r - x*x
            fun above y = (* 大于或等于y的所有序偶 *)
                  if y>x then []
                  else if y*y<diff then above (y+1)
                  else if y*y=diff then (x,y)::between(x-1,y+1)
                  else (* y*y>diff *) between(x-1,y)
            in above y end;
        val firstx = floor(Math.sqrt(real r))
      in between (firstx, 0) end;
  > val squares = fn : int -> (int*int) list
```

执行速度很快，即使 r 是很大的数：

```
squares 50;
> [(7, 1), (5, 5)] : (int*int) list
squares 1105;
```

① 这是一个困难的问题：组件 *zero* 是什么？一个显然的选择就是 $[]$ ，但是矩阵的操作都需要小心修改，以便正确地将其处理作零矩阵。

```
> [(33, 4), (32, 9), (31, 12), (24, 23)] : (int*int) list
squares 48612265;
> [(6972, 59), (6971, 132), (6952, 531), (6948, 581),
> (6944, 627), (6917, 876), (6899, 1008), (6853, 1284),
> (6789, 1588), (6772, 1659), ...] : (int*int) list
```

Dijkstra的程序使用了另外一种搜索方法： x 和 y 从相等的值开始分别向两头扫描。我们在这里所用的推导方法很可能是他不想要的，因为“想说明没有漏掉任何一个解，少不了要画张图吧。”^①

❶ 更聪明的方法？一个数可以恰为两个数的平方和，仅当它的质因数分解里，所有的形如 $4k+3$ 的因子的指数都是偶数。例如， $48\ 612\ 265 = 5 \times 13 \times 17 \times 29 \times 37 \times 41$ ，这些质数没有一个是形如 $4k+3$ 的。条件本身主要告诉我们解是否存在，不过这个理论也提供了一种枚举解的方法（Davenport, 1952, 第V章）。只有计算很大的数时，利用这一理论的程序才是值得的。

94

3.13 求后继排列的问题

已知一列整数，要将它们重新排列形成按字典顺序的后继排列。新的排列要比原来的大，并且没有其他排列夹在两者之间。

我们对这个问题做一点修改。字典顺序的意思是表头元素是最高位，而后继的排列很可能只在最低位元素上有所区别。由于表头元素最容易访问，因此将表头元素作为最低位，我们也没有必要在是否依表的自然顺序这个问题上争论。这样，就按逆字典顺序来计算后继排列。

这个问题很难形象地描述，即使是Dijkstra也需要举个例子。下面是4 3 2 1（最初的排列）之后的8个排列：

```
3 4 2 1
4 2 3 1
2 4 3 1
3 2 4 1
2 3 4 1
4 3 1 2
3 4 1 2
4 1 3 2
```

每一个排列中受影响的部分都由下划线标出。排列的序列在1 2 3 4处终止，它没有后继。

要求得更大的排列，表中的某些元素必定要被它左侧的一个更大的元素所替换。要求得恰好下一个排列，这个被替换的元素必须尽量靠左，也就是尽量靠低位。用来替换的元素也必须要尽量小，并且被替换的元素的左侧也要重新从左到右按降序排列。所有这些可以通过下面两步完成：

(1) 找到最左边的一个元素 y ，使得其左侧存在比它大的元素。显然，这个元素的左侧的

① 这是作者的幽默。我不知道Dijkstra有没有说过这句话，这位大师已于2002年去世了。作为令人敬仰的学者，他为计算机科学作出了丰富而深刻的贡献，同时也留下了个人严谨而有时近乎怪癖的观点和风格。——译者注

那些元素是按升序排列的序列 $x_1 \leq \dots \leq x_n$ 。(这里我们实际是用元素的位置来描述元素,不过这只在元素彼此不同的时候才是可以的。)

(2) 将 y 替换成最小的 x_i , 满足 $1 \leq i \leq n$ 且 $y < x_i$, 并将序列 $x_1, \dots, x_{i-1}, y, x_{i+1}, \dots, x_n$ 重新按降序排列。要完成这个任务, 可以扫描序列 x_n, x_{n-1}, \dots, x_1 直到发现正确的 x_i , 并将较大的值放在结果的前面。

调用函数`next(xlist, ys)`可以在`ys`里面找到需要替换的 y , 同时`xlist`累积了那些被略过的元素。当`xlist`存放了那个逆向的序列 $[x_n, \dots, x_1]$ 后, 函数`swap`就完成了替换和重新按降序排列的任务。这里, 表的操作是很精致的。

```
fun next(xlist, y::ys) : int list =
  if hd xlist <= y then next(y::xlist, ys)
  else (* 将y和满足x >= xk > y的最大的xk交换 *)
    let fun swap [x] = y::x::ys
        | swap (x::xk::xs) = (*x >= xk*)
          if xk > y then x::swap(xk::xs)
          else (y::xk::xs)@(x::ys)
          (*x > y >= xk >= xs*)
        in swap(xlist) end;
  > val next = fn : int list * int list -> int list
```

函数`nextperm`启动这个扫描。

```
fun nextperm (y::ys) = next([y], ys);
> val nextperm = fn : int list -> int list
nextperm [1,2,4,3];
> [3, 2, 1, 4] : int list
nextperm it;
> [2, 3, 1, 4] : int list
nextperm it;
> [3, 1, 2, 4] : int list
```

这对于有相同元素的表也适用:

```
nextperm [3,2,2,1];
> [2, 3, 2, 1] : int list
nextperm it;
> [2, 2, 3, 1] : int list
nextperm it;
> [3, 2, 1, 2] : int list
```

练习 3.29 写出`nextperm[2,3,1,4]`的计算步骤。

练习 3.30 如果`next`函数第二行的`<`换成了`<=`, 那么这个函数还对吗? 根据上面所讨论的那两个步骤来给出你的答案。

练习 3.31 如果`ys`没有后继排列, 那么`nextperm(ys)`返回什么? 修改程序使得它在这种情况下返回初始排列(字典顺序中最小的排列)。

多态函数中的相等测试

像`length`和`rev`那样的多态函数可以接受具有任何类型元素的表, 因为它们对表中的元素并不进行任何操作。现在来看一个函数, 它测试一个值 e 是否存在于一个表 l 中, 这个函数是多

态的吗？每个 l 中的元素都要和 e 进行相等测试。相等测试是多态的，不过有一定的限制。

3.14 相等类型

相等类型 (equality type) 是一种其值允许相等测试的类型。相等测试对于函数类型和抽象类型都是行不通的：

- 对于两个函数的相等测试是不可计算的，因为 f 和 g 是相等的仅当对于每一个可能的 x ， $f(x)$ 等于 $g(x)$ 。也有其他的方法来定义函数的相等，但都逃不过这个问题。
- 抽象类型只提供在其定义中所指定的操作。ML隐藏了具体的类型表达实体的相等测试，因为它往往和所期望的抽象意义上的相等并不一致。[⊖]

相等对于许多基本类型是有定义的：整数、实数、字符、字符串和布尔类型。对于结构类型的值，相等测试比较它们的相应分量，这样一来，相等测试对于由基本类型组成的元组、记录、表和数据类型 (datatype, 将在下一章介绍) 都有定义。而在那些包含函数或抽象类型元素的值上并没有定义相等测试。

Standard ML提供了相等类型变量 (equality type variable) $\alpha^=$ 、 $\beta^=$ 、 $\gamma^=$ 、 \dots ，它们在相等类型的范围内变化。(多态的)相等类型不包括相等类型变量以外的类型变量。例如， int 、 $bool \times string$ 和 $(int\ list) \times \beta^=$ 都是相等类型，而 $int \rightarrow bool$ 和 $bool \times \beta^=$ 则不是。

下面是相等测试中缀操作符 ($=$) 本身的类型：

```
op= ;
> fn : ('a * 'a) -> bool
```

这个类型的数学记法是 $\alpha^= \times \alpha^= \rightarrow bool$ 。在ML中，一个相等类型变量的名字由两个撇号 (') 字符开始。

我们来声明一下成员测试函数：

```
infix mem;
fun (x mem []) = false
  | (x mem (y::l)) = (x=y) orelse (x mem l);
> val mem = fn : 'a * 'a list -> bool
```

类型 $\alpha^= \times (\alpha^= \text{ list}) \rightarrow bool$ 表示了 mem 可以应用到所有元素类型为相等类型的表上去。

```
"Sally" mem ["Regan", "Goneril", "Cordelia"];
> false : bool
```

3.15 多态集合操作

如果一个函数进行了多态的相等测试，即使是间接的，例如通过函数 mem ，那么这个函数的类型里面就包含有相等类型变量。函数 $newmem$ 可以往一个表里面添加一个新元素，并保证确实是新的 (原来没有的)。

```
fun newmem(x,xs) = if x mem xs then xs else x::xs;
> val newmem = fn : 'a * 'a list -> 'a list
```

通过 $newmem$ 构造的表可以被看作是有限集合。[⊕]让我们来声明一些集合的操作，并注意一下

⊖ 第7章详细介绍了抽象类型。

⊕ 后面的3.22节更深入地讨论了用一种数据结构来表示另一种数据结构的问题。

它们的类型。如果出现了相等类型变量，那么就说明涉及了相等测试。

函数`setof`通过删去重复元素来将一个表转换为“集合”：

```
fun setof [] = []
  | setof (x::xs) = newmem(x, setof xs);
> val setof = fn : 'a list -> 'a list
setof [true,false,false,true,false];
> [true, false] : bool list
```

看得出`setof`可能会进行非常多的相等测试。为了减少`setof`的使用，下面的函数可以被应用到没有重复元素的表所表示的所谓“集合”上，并保证它们的结果也是一个“集合”。

并集。`union(xs, ys)`返回的表加进了所有`xs`中没有出现在`ys`中的元素，假设`ys`已经是一个没有重复元素的表：

```
fun union([],ys) = ys
  | union(x::xs, ys) = newmem(x, union(xs, ys));
> val union = fn : 'a list * 'a list -> 'a list
```

类型变量`a`表明使用了相等测试，在这里是通过`newmem`进行的。

```
union([1,2,3], [0,2,4]);
> [1, 3, 0, 2, 4] : int list
```

98

交集。类似地，`inter(xs, ys)`包括了所有`xs`和`ys`共有的元素：

```
fun inter([],ys) = []
  | inter(x::xs, ys) = if x mem ys then x::inter(xs, ys)
                        else inter(xs, ys);
> val inter = fn : 'a list * 'a list -> 'a list
```

婴儿的名字可以在父母喜欢的名字的交集里面选择

```
inter(["John", "James", "Mark"], ["Nebuchadnezzar", "Bede"]);
> [] : string list
```

……虽然这个办法很少管用。

子集关系。当集合`T`的所有元素都是集合`S`的元素时，`T`就是`S`的子集 (subset)：

```
infix subs;
fun ([] subs ys) = true
  | ((x::xs) subs ys) = (x mem ys) andalso (xs subs ys);
> val subs = fn : 'a list * 'a list -> bool
```

记得相等类型也包括元组、表等：

```
[("May",5), ("June",6)] subs [("July",7)];
> false : bool
```

集合的相等。内置的表的相等测试对集合是无效的。表`[3, 4]`和表`[4, 3]`是不等的，但它们表示了同一个集合。集合的相等是忽略顺序的。它可以用子集来定义：

```
infix seq;
fun (xs seq ys) = (xs subs ys) andalso (ys subs xs);
> val seq = fn : 'a list * 'a list -> bool
[3,1,3,5,3,4] seq [1,3,4,5];
> true : bool
```

集合应该被定义成抽象类型，将表的相等测试隐藏起来。

幂集。集合 S 的幂集 (powerset) 是所有 S 的子集所组成的集合，包括空集合和 S 本身。它可以通过计算从 S 中移去一个元素 x 后的集合 $S-\{x\}$ 的幂集来递归求得。如果集合 T 是 $S-\{x\}$ 的子集，那么 T 和 $T \cup \{x\}$ 两者都是 S 的子集，也就是 S 幂集的元素。参数 $base$ 积累那些必须包含在结果幂集的每一个元素集合中的，像 x 那样的元素。在一开始， $base$ 必须为空。

99

```
fun powset ([], base) = [base]
  | powset (x::xs, base) =
    powset(xs, base) @ powset(xs, x::base);
> val powset = fn : 'a list * 'a list -> 'a list list
```

这里，普通的类型变量表明了 $powset$ 并没有进行相等测试。

```
powset (rev ["the","weird","sisters"], []);
> [[], ["the"], ["weird"], ["the", "weird"], ["sisters"],
> ["the", "sisters"], ["weird", "sisters"],
> ["the", "weird", "sisters"]] : string list list
```

使用集合的记法， $powset$ 的结果可以像下面那样描述，忽略掉表元素的顺序：

$$powset(S, B) = \{T \cup B \mid T \subseteq S\}$$

笛卡尔乘积。集合 S 和集合 T 的笛卡尔乘积 (Cartesian product) 就是所有满足 $x \in S$ 且 $y \in T$ 的序偶 (x, y) 的集合。用集合的记法就是：

$$S \times T = \{(x, y) \mid x \in S, y \in T\}$$

好几种函数式语言都支持一些集合记法，这是追随了David Turner，范例详见Bird和Wadler (1988)。由于ML并不支持，我们必须在表上使用递归。计算笛卡尔乘积的函数出乎意料地复杂。

```
fun cartprod ([], ys) = []
  | cartprod (x::xs, ys) =
    let val xsprod = cartprod(xs, ys)
        fun pairx [] = xsprod
          | pairx(y::ytail) = (x,y) :: (pairx ytail)
        in pairx ys end;
> val cartprod = fn : 'a list * 'b list -> ('a * 'b) list
```

函数 $cartprod$ 并不进行相等测试。

```
cartprod([2,5], ["moons","stars","planets"]);
> [(2, "moons"), (2, "stars"), (2, "planets"),
> (5, "moons"), (5, "stars"), (5, "planets")]
> : (int * string) list
```

5.10节会展示怎样用高阶函数来表示这个函数。目前，让我们继续使用简单的方法。

练习 3.32 在计算下面的表达式时，ML需要进行多少次相等测试？

```
1 mem upto(1,500)
setof(upto(1,500))
```

100

练习 3.33 比较函数 $union$ 和下面定义的 $itunion$ 。哪个函数效率更高？

```
fun itunion([],ys)      = ys
  | itunion(x::xs,ys) = itunion(xs,newmem(x,ys));
```

练习 3.34 书写函数`choose`，使得`choose(k, xs)`产生集合`xs`的所有`k`个元素的子集合所组成的集合。例如，`choose(29, upto(1, 30))`应该返回一个包括30个子集合的表。

练习 3.35 下面的函数比`cariprod`要简单。它会更好地计算笛卡儿乘积吗？

```
fun cprod([],ys) = []
  | cprod(x::xs,ys) =
    let fun pairx [] = cprod(xs,ys)
        | pairx(y::ytail) = (x,y) :: (pairx ytail)
    in pairx ys end;
```

3.16 关联表

字典或表格可以用序偶的表来表示。搜索这种表格的函数涉及到相等测试。要存放历史上重大战役的时间，我们可以这样写

```
val battles =
  [("Crecy",1346), ("Poitiers",1356), ("Agincourt",1415),
   ("Trafalgar",1805), ("Waterloo",1815)];
```

(键, 值)的序偶表叫做关联表 (association list)。函数`assoc`通过顺序搜索来查找与一个键关联的值:

```
fun assoc([],a)      = []
  | assoc((x,y)::pairs,a) = if a=x then [y]
                             else assoc(pairs,a);
> val assoc = fn : ('a * 'b) list * 'a -> 'b list
```

它的类型 $(\alpha \times \beta)list \times \alpha \rightarrow \beta list$ ，说明了键必须具有某种相等类型 α ，而值则可以使用任何类型 β 。调用`assoc(pairs, x)`返回 `[]` 则说明键`x`没有找到，如果找到与`x`配对的`y`，则会返回`[y]`。把结果作为表来返回是一种简单的区分成功和失败的方法。

101

```
assoc(battles, "Agincourt");
> [1415] : int list
assoc(battles, "Austerlitz");
> [] : int list
```

搜索可能会很慢，但是更新却很快，只要把新的序偶放到前面就行了。由于`assoc`返回它所发现的第一个值，所以旧的就被新的覆盖了。在块式结构化语言中将名字和它的类型关联起来就是关联表的一种典型应用。如果一个名字在嵌套的多个块中都有声明，那么这个名字可能以不同类型多次出现在一个关联表中。

❶ 相等类型：好还是坏？ Appel (1993) 基于多个方面对ML的相等多态性进行了批评。这使得语言的定义更复杂了。它同时也使实现更复杂了：数据必须具有运行时的标记来支持相等测试，否则一个相等测试的方法就要隐式地传入用到它的那些函数中去。有时，标准的相等测试是不恰当的，例如在集合的集合这种情况下。多态的相等测试也会很慢。

使用相等多态性的部分原因是历史性的。ML和Lisp很有关系，在Lisp中像`mem`

和`assoc`这样的函数都作为基本原语。但是，就连Lisp也必须对这些函数提供不同的版本进行不同类型的相等测试。如果ML没有相等多态性，那些函数仍然可以通过传入额外的测试函数来表达。

相等实际上是被重载了的：它的含义取决于它的类型。其他的重载函数包括那些算术运算以及将值表示为字符串的函数。ML对重载的处理看来很不令人满意，特别是和Haskell典雅的类型类（type class）（Hudak等，1992）相比。但是类型类也令语言更加复杂化。更严重的是，一个程序在没有彻底地类型检查之前根本不能运行，哪怕只是原则上的。Odersky等（1995）讨论了另一种替代的设计；这方面需要更多的研究。

3.17 图的算法

序偶表也可以用来表示有向图。每个序偶 (x, y) 代表一条 $x \rightarrow y$ 的边。这样，表

```
val graph1 = [("a", "b"), ("a", "c"), ("a", "d"),
              ("b", "e"), ("c", "f"), ("d", "e"),
              ("e", "f"), ("e", "g")];
```

就表示了图3-1a所示的图。

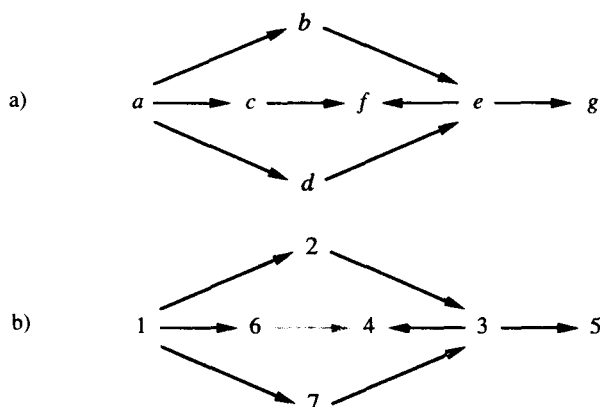


图3-1 有向图和深度优先遍历

函数`nexts`从图中找出结点`a`的所有后继，也就是从`a`出发的边的终点：

```
fun nexts (a, []) = []
  | nexts (a, (x,y)::pairs) =
    if a=x then y :: nexts(a,pairs)
    else      nexts(a,pairs);
> val nexts = fn : 'a * ('a * 'b) list -> 'b list
```

这个函数和`assoc`有点不同，它返回所有与`a`组成序偶的值，而不仅是第一个值：

```
nexts("e", graph1);
> ["f", "g"] : string list
```

深度优先搜索。许多图的算法都是顺着边访问结点，并记住那些访问过的结点，这样每个结点最多访问一次。在深度优先搜索（depth-first search）中，从当前结点可以到达的子图

将首先被浏览, 然后才轮到其他结点。函数`depthf`实现了这一搜索策略, 利用参数`visited`来将访问过的结点按逆序积累起来:

```
fun depthf ([], graph, visited) = rev visited
  | depthf (x::xs, graph, visited) =
    if x mem visited then depthf (xs, graph, visited)
    else depthf (nexts(x, graph) @ xs, graph, x::visited);
> val depthf = fn
> : 'a list * ('a * 'a) list * 'a list -> 'a list
```

图的结点可以是任何的相等类型。

从`a`开始的`graph1`的深度优先搜索按图3-1b所示的顺序访问诸结点。有一条边没有遍历到。

103

```
depthf(["a"], graph1, []);
> ["a", "b", "e", "f", "g", "c", "d"] : string list
```

添加一条从`f`到`d`的边使得该图有了一个圈。如果从结点`b`开始搜索图, 那么圈的一条边会被忽略。另外, 图的一部分由`b`根本访问不到; 见图3-2。

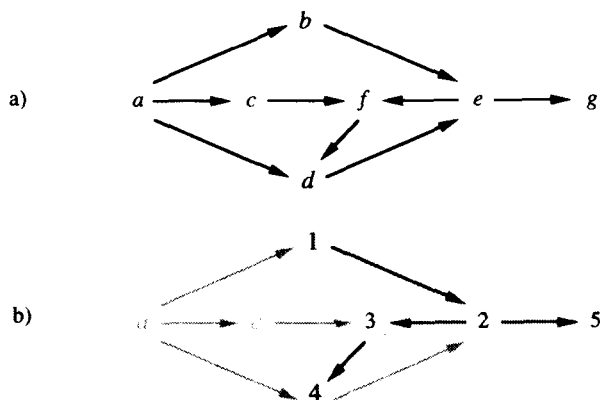


图3-2 循环图和深度优先遍历

```
depthf(["b"], ("f","d")::graph1, []);
> ["b", "e", "f", "d", "g"] : string list
```

当访问了一个之前没有访问过的结点`x`之后, 深度优先搜索递归地访问每一个`x`的后继。在通过`nexts(x, graph) @ xs`计算出的表中, `x`的后继都被放在了其他等待访问的结点列表`xs`之前。这个表的作用像一个栈。如果这个表被作为一个队列的话, 我们就得到了广度优先搜索 (breadth-first search)。

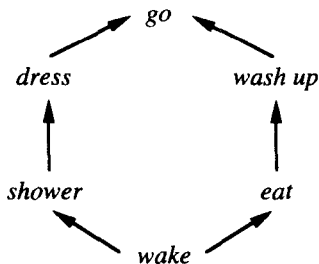
104

深度优先搜索也可以如下面这样编写:

```
fun depth args =
  let fun rdepth ([], graph, visited) = visited
        | rdepth (x::xs, graph, visited) =
            rdepth (xs, graph,
              if x mem visited then visited
              else rdepth (nexts(x, graph), graph, x::visited))
  in rev(rdepth args) end;
```

嵌套的递归调用先访问了 x 的后继，然后另外一个递归调用访问其他的结点， xs 。由于存在内层递归，因此不能像 $depthf$ 的迭代式递归那样直接返回 $rev\ visited$ ，那样会把内层递归的中间结果也翻转了。我们利用 $rdepth$ 来返回逆向结果，最后再整体做一次翻转。两个函数 $depthf$ 和 $depth$ 是等价的，虽然证明这个需要一定的技巧。由于少用了一个追加调用($@$)， $depth$ 会快一点。更重要的是，由于一个调用专门用来访问结点 x ，可以很容易地将它修改以检测图中的圈，或进行拓扑排序。

拓扑排序。事件发生顺序的约束形成了一个图。每一条边 $x \rightarrow y$ 代表了“ x 必须在 y 之前发生。”下图



描述了上班之前需要做的事情。下面是描述这个图的代码列表：

```
val grwork = [("wake", "shower"), ("shower", "dress"),
              ("dress", "go"),      ("wake", "eat"),
              ("eat", "washup"),    ("washup", "go")];
```

从图中找到一个线性的事件序列被称为拓扑排序 (topological sorting)。Sedgewick (1988) 指出，深度优先搜索可以完成这项任务，要求是在搜索到 x 的后继之后，将 x 记录下来。这样一来 x 就出现在它所能到达的所有结点之后：反向的拓扑排序。

这意味着对 $depth$ 要进行简单的修改：把 x 放在递归调用的结果之前，而不是它的参数 $visited$ 之前。这个表本来就是倒着生成的，所以不需要再翻转了。

105

```
fun topsort graph =
  let fun sort ([], visited) = visited
      | sort (x::xs, visited) =
          sort(xs, if x mem visited then visited
                  else x :: sort(nexts(x, graph), visited))
      in
        val (starts, _) = ListPair.unzip graph
        sort(starts, [])
      end;
  > val topsort = fn : ('a * 'a) list -> 'a list
```

let 声明让 $sort$ 可以引用 $graph$ ，它也声明了所有边的开始结点 $starts$ ，这保证了图的所有结点都能被访问到。

那么，我们上班前要怎样进行准备呢？

```
topsort grwork;
> ["wake", "eat", "washup", "shower", "dress", "go"]
> : string list
```

翻转表示有向边的表会给我们带来不同的答案：

```
topsort(rev grwork);
> ["wake", "shower", "dress", "eat", "washup", "go"]
> : string list
```

图的检测。现在考虑多一个约束：我们必须在“吃”(eat)之前“走”(go)。得到的图里面有一个圈，没有任何(拓扑排序的)解。下面的函数调用将永远运行下去：

```
topsort(("go", "eat")::grwork);
```

死循环是不能接受的；因此函数应该以某种方式报告没有解。圈可以通过维护一个正在搜索的所有结点的表来检测，这个表叫做路径`path`，它从搜索一开始就跟踪经过的边。

```
fun pathsort graph =
  let fun sort ([], path, visited) = visited
      | sort (x::xs, path, visited) =
          if x mem path then hd[] (*abort!!*)
          else sort(xs, path,
                    if x mem visited then visited else
                    x :: sort(nexts(x, graph), x::path, visited))
      val (starts, _) = ListPair.unzip graph
  in sort(starts, [], []) end;
> val pathsort = fn : ('a * 'a) list -> 'a list
```

对于原来的图这个函数照样工作，而对于含有圈的图，它会产生错误：

```
pathsort graph1;
> ["a", "d", "c", "b", "e", "g", "f"] : string list
pathsort(("go", "eat")::grwork);
> Exception: Match
```

错误信息要比死循环好得多，但是`pathsort`是通过一个错误的函数调用(`hd []`)来终止运行的，这是一个不怎么样的技巧。下一章会解释怎样为这种错误声明一个异常(exception)。

异常并不是报告存在圈的唯一办法。下面的函数返回两个结果：访问过的结点的表，这跟以前一样，以及在圈中所发现的结点的表。为了维护两个结果，让我们先声明一个函数来增加访问过的结点：

```
fun newvisit (x, (visited, cys)) = (x::visited, cys);
> val newvisit = fn : 'a * ('a list * 'b) -> 'a list * 'b
```

有了这个函数，表示拓扑排序就很容易了：

```
fun cyclesort graph =
  let fun sort ([], path, (visited, cys)) = (visited, cys)
      | sort (x::xs, path, (visited, cys)) =
          sort(xs, path,
              if x mem path then (visited, x::cys)
              else if x mem visited then (visited, cys)
              else newvisit(x, sort(nexts(x, graph),
                                    x::path, (visited, cys))))
      val (starts, _) = ListPair.unzip graph
  in sort(starts, [], ([], [])) end;
> val cyclesort = fn
> : ('a * 'a) list -> 'a list * 'a list
```

如果存在圈，`cyclesort`会报告圈在什么地方：

```
cyclesort (("go", "eat")::grwork);
> ([ "wake", "shower", "dress", "go", "eat", "washup"],
>  ["go"]) : string list * string list
```

如果没有, *cyclesort*便对图进行排序:

```
cyclesort(rev graph1);
> ([ "a", "b", "c", "d", "e", "f", "g"], [])
> : string list * string list
```

这些多态的图函数对于大图是很慢的, 因为进行了很多表的搜索。如果将结点限制在整数类型上, 则可以利用下一章的函数式数组来写出效率更高的函数。

练习 3.36 修改*pathsort*, 当图含有圈时返回 [], 否则返回单元元素表 [*visited*]。

练习 3.37 设(*visited*, *cys*)是*cyclesort*的结果。如果图里面有多个圈, 每个圈都会有一个结点在*cys*里面吗? 如果图里面有圈, *visited*里面返回的是什么呢?

107

排序: 案例研究

排序是计算理论中研究最多的题目之一。有些排序算法非常出名。要将 n 个元素排序, 插入排序 (insertion sort) 需要 $O(n^2)$ 时间; 合并排序 (merge sort) 需要 $O(n \log n)$ 时间; 快速排序 (quick sort) 平均需要 $O(n \log n)$ 时间, 而最坏情况则是 $O(n^2)$ 。

这些算法通常是对数组进行排序。除了堆排序 (heap sort), 它用数组来编码存放一棵二叉树, 其他的都可以很容易地写成作用于表的函数。它们的时间复杂度也保持不变: 并不是说表的排序会在速度上胜过数组! $O(n^2)$ 的时间复杂度估计是指执行时间和 n^2 成正比。而表的排序会有较大的比例系数。

这一节比较了几种排序算法, 给出了将10 000个随机数排序的时间。这些时间并不是正式的, 但也体现出各种算法实践上的性能。

Sedgewick (1988) 所写的Pascal版本的快速排序可以在110毫秒内完成排序。这与函数式排序的最好时间相仿。Pascal在一些检测被省掉后要胜过ML, 但是却牺牲了函数式程序设计的清晰和简洁, 更不用说安全了。表的开销在排序里面并不很重要, 比如排序一个参考文献表, 时间主要是花在比较上面了。

❶ 时间是怎样测量的。时间测量是在一台Sun SuperSPARC Model 61计算机上进行的, 运行的是Standard ML of New Jersey, version 108。测量利用了标准库中的设施 (Timer结构), 并且包括了垃圾收集的时间。感谢软件和硬件的进步, ML程序比写这本书第一版的时候快了20~40倍。

Pascal程序是使用Pascal 3.0编译器编译的。不使用数组下标检测的话, 时间下降到了75毫秒。如果使用完全优化的话, 程序只需34毫秒便完成了, 但是之后打出了一个警告消息: 它“可能输出了非标准的浮点数结果”。在追求速度的道路上, 我们值得冒什么样的风险呢?

3.18 随机数

首先我们必须产生10 000个随机数。Park和Miller (1988), 在抱怨很难找到好的随机数

108 发生器的情况下，推荐了下面的算法：

```
local val a = 16807.0 and m = 2147483647.0
in fun nextrand seed =
    let val t = a*seed
    in t - m * real(floor(t/m)) end
end;
> val nextrand = fn : real -> real
```

调用nextrand，传入从1到m-1之间的任何seed就会产生在此范围内的另一个数，它实际上进行了下面的整数运算

$$(a \times seed) \bmod m$$

使用实数是为了避免整数溢出。只要实数尾数足够46位，这个函数就能正确运行。当在你的机器上试验的时候，要检查一下产生的随机数是否恰好是整数。

调用函数randlist(n, seed, [])产生一个从seed开始的具有n个元素的随机数表。由于表是在tail中积累的，因此它的顺序是倒的：

```
fun randlist (n, seed, tail) =
    if n=0 then (seed, tail)
    else randlist(n-1, nextrand seed, seed::tail);
> val randlist = fn
> : int * real * real list -> real * real list
```

10 000个随机数的表叫做rs。下面显示了头15个。

```
val (seed, rs) = randlist(10000, 1.0, []);
> val seed = 1043618065.0 : real
> val rs =
> [1484786315.0, 925166085.0, 1614852353.0, 721631166.0,
> 173942219.0, 1229443779.0, 789328014.0, 570809709.0,
> 1760109362.0, 270600523.0, 2108528931.0, 16480421.0,
> 519782231.0, 162430624.0, 372212905.0, ...] : real list
```

3.19 插入排序

插入排序是将元素一次一个地插入到已经排序的表中。它很慢，不过简单。下面是插入函数：

```
fun ins (x, []): real list = [x]
  | ins (x, y::ys) =
    if x<=y then x::y::ys (* 它属于这里 *)
    else y::ins(x, ys);
> val ins = fn : real * real list -> real list
```

类型约束real list解决了比较操作符的重载问题。所有排序函数都有类型约束。

109 我们向显然是有序的表[6.0]中插入一些数：

```
ins(4.0, [6.0]);
> [4.0, 6.0] : real list
ins(8.0, it);
> [4.0, 6.0, 8.0] : real list
ins(5.0, it);
> [4.0, 5.0, 6.0, 8.0] : real list
```

插入排序函数对每一个输入的元素调用`ins`:

```
fun insert [] = []
  | insert (x::xs) = ins(x, insert xs);
> val insert = fn : real list -> real list
```

这些函数都需要深层递归, 不过这方面的低效并不重要。插入, 无论是函数式的还是命令式的, 都需要做大量的复制。运行时间和 n^2 成正比。对于我们的10 000个数来说是超过32秒, 大约比快速排序慢300倍。插入排序只能考虑对短表使用, 或对于几乎排好的表使用。这个算法还是值得注意的, 因为它很简单, 也因为它的基础上, 能提炼出更好的算法(合并排序和堆排序)。

3.20 快速排序

由C. A. R. Hoare发明的快速排序是最快的排序之一。它的原理是分治法:

- 从输入中选择某个值 a , 称为主元(pivot)。
- 将剩下的项目分为两部分: 一部分小于或等于 a , 另一部分大于 a 。
- 分别递归地排序两个部分, 并将小的那部分放在大的前面。

快速排序对于数组是很理想的, 分区操作极快, 只需要移动很少的项。对于表, 分区需要复制所有的项; `partition`是一个不错的迭代函数的例子, 它构造出两个结果。

```
fun quick [] = []
  | quick [x] = [x]
  | quick (a::bs) = (* 表头"a"是主元 *)
    let fun partition (left, right, []): real list =
          (quick left) @ (a :: quick right)
        | partition (left, right, x::xs) =
          if x <= a then partition (x::left, right, xs)
          else partition (left, x::right, xs)
        in partition([], [], bs) end;
> val quick = fn : real list -> real list
```

这个函数将我们的10 000个数进行排序, 用了大约160毫秒:

```
quick rs;
> [1.0, 8383.0, 13456.0, 16807.0, 84083.0, 86383.0,
> 198011.0, 198864.0, 456291.0, 466696.0, 524209.0,
> 591308.0, 838913.0, 866720.0, ...] : real list
```

110

通过使用第二个参数来积累结果, 追加操作(`@`)是可以避免的。这个版本的快速排序留作练习, 它只需要110毫秒。

和相应的过程式程序一样, `quick`平均需要和 $n \log n$ 成正比的时间。如果输入已经是升序或者降序的话, 快速排序则需要和 n^2 成正比的时间。

练习 3.38 重新表达快速排序, 使得`quicker(xs, sorted)`通过`sorted`来积累结果, 从而省去追加操作。

练习 3.39 书写函数`find`, 使得`find(xs, i)`返回表`xs`中第 i 小的元素。这被称为选择(selection)。Hoare的算法和快速排序有关, 并且以比排序快得多的速度返回第 i 个元素。

练习 3.40 推广`find`, 使得`findrange(xs, i, j)`返回表`xs`中第 i 小到第 j 小之间的元素列表。

3.21 合并排序

有几个算法是基于合并已排序的表来工作的。合并函数不断地从两个表中提取较小的那个表头元素：

```
fun merge([],ys)      = ys : real list
  | merge(xs,[])      = xs
  | merge(x::xs, y::ys) =
      if x<=y then x::merge(xs, y::ys)
      else y::merge(x::xs, ys);
> val merge = fn : real list * real list -> real list
```

当排序10 000个项目时，*merge*中的递归对于有些ML系统来说是过深了。这是那些ML系统的问题，而不是*merge*的问题。像*take*和*append*一样，主要的开销是用于构造结果。迭代的合并函数，虽然避免了深层的递归，但是却很可能必须进行昂贵的翻转操作。

合并排序可以是自顶向下的（top-down），也可以是自底向上的（bottom-up）。不论哪一种，合并只有在两个表的长度差不多的时候才是有效的。如果其中一个表只有一个元素，那么合并就退化成了插入。

自顶向下的合并排序。在自顶向下的方法中，输入的表利用*take*和*drop*被分成长度大致相等的两部分。对它们分别递归地进行排序，然后把结果合并在一起。

111

```
fun tmergesort [] = []
  | tmergesort [x] = [x]
  | tmergesort xs =
      let val k = length xs div 2
      in merge (tmergesort (List.take(xs,k)),
                tmergesort (List.drop(xs,k)))
      end;
> val tmergesort = fn : real list -> real list
```

不像快速排序，合并排序在最坏情况下需要的时间仍和 $n \log n$ 成正比。但是平均来说，它比较慢，需要290毫秒来排序10 000个数。它对于*length*、*take*和*drop*的调用重复地扫描了输入的表。下面是减少重复扫描的一种办法：

```
fun mergesort' xs =
  let fun sort (0, xs) = ([], xs)
      | sort (1, x::xs) = ([x], xs)
      | sort (n, xs) =
          let val (l1, xs1) = sort ((n+1) div 2, xs)
              val (l2, xs2) = sort (n div 2, xs1)
          in (merge (l1,l2), xs2)
          end
      val (l, _) = sort (length xs, xs)
  in l end;
> val mergesort = fn : real list -> real list
```

调用*sort*(*n*, *xs*)将*xs*的前*n*个元素排序，并将剩下元素的一起返回（在结果的第二个分量中）。有人可能会觉得它慢，因为它构造了很多的序偶。但是这个函数只需要200毫秒来排序我们的随机数表。虽然它仍然比快速排序要慢，但是已经可以作为简单而快速的排序方法，推荐给大家了。

自底向上的合并排序。基本的自底向上方法是将整个输入表分割成长度为1的表。然后靠

在一起的表就可以被合并了，得到有序的长度为2的表，然后是长度为4的表，然后是长度为8的表，依此类推。最后就只剩下一个有序的表了。这个方案很容易编码，但是非常浪费。为什么10 000个数的表要被复制成10 000个单元元素表呢？

O'Keefe (1982) 描述了一个漂亮的方法来同时合并不同长度的表，而无须完整地存储这些表。

ABCDEFGHIJK

下划线显示了相邻的表示是怎样被合并的。首先是A和B，然后是C和D，现在AB和CD拥有相同的长度，于是又可以合并了。O'Keefe加快了将所有层次的合并积累在一个表中的速度。它不去比较两个表的长度，而是让成员的计数 k 去决定如何加入下一个成员。如果 k 是偶数，那么说明有两个长度同为 s 的成员需要合并。合并之后的表作为具有长度 $2s$ 的成员 $k/2$ ，它可能引起进一步的合并。

112

```
fun mergepairs([l], k) = [l]
  | mergepairs(l1::l2::ls, k) =
    if k mod 2 = 1 then l1::l2::ls
    else mergepairs(merge(l1,l2)::ls, k div 2);
> val mergepairs = fn
> : real list list * int -> real list list
```

如果 $k = 0$ ，那么mergepairs就将整个表的表合并成一个表。调用sorting(xs, [], 0)排序表xs。它需要270毫秒来排序10 000个随机数。

```
fun sorting([], ls, k) = hd(mergepairs(ls, 0))
  | sorting(x::xs, ls, k) =
    sorting(xs, mergepairs([x]::ls, k+1), k+1);
> val sorting = fn
> : real list * real list list * int -> real list
```

k 可以理解为一个计数器，是当前加入合并的输入子表数目（这里是单元元素表，但不局限于此）。如果把 k 写成二进制数，就可以体会mergepairs的巧妙之处。事实上mergepairs相当于二进制加法的进位过程，在 k 不为0时，把 k 个输入子表合并成 k 的二进制表示中1的个数那么多个子表。如果 k 加一以后导致二进制表示的右端出现0，那么有多少个连续的0，原先 k 的右端就有多少个连续的1，也就合并这么多个子表成为一个新的子表。——译者注

平滑（smooth）排序在输入基本有序的情况下具有线性（和 n 成正比）的时间复杂度，而在最坏的情况下退化回 $O(n \log n)$ 。O'Keefe利用输入的有序性，展示了一种“平滑应用式合并排序”。他把输入分成一个个自身升序的行程，来取代单元元素表。如果行程的数目和 n 无关（也就是“基本有序”），那么运行时间就是线性的。

函数nextrun从表中取得下一个升序的行程，并与剩下的，没有读到的项所组成的表配到一起返回。（命令式程序会将处理过的项删除。）行程表是反向增长的，所以要调用rev。

```
fun nextrun(run, []) = (rev run, []: real list)
  | nextrun(run, x::xs) =
    if x < hd run then (rev run, x::xs)
    else nextrun(x::run, xs);
> val nextrun = fn
> : real list * real list -> real list * real list
```


行程表不断地被取出和合并的。

```
fun samsorting([], ls, k) = hd(mergepairs(ls, 0))
  | samsorting(x::xs, ls, k) =
    let val (run, tail) = nextrun([x], xs)
    in samsorting(tail, mergepairs(run::ls, k+1), k+1)
    end;
> val samsorting = fn
> : real list * real list list * int -> real list
```

113 最后，主排序函数为

```
fun samsort xs = samsorting(xs, [[]], 0);
> val samsort = fn : real list -> real list
```

这个算法既典雅又高效。即使是对于短行程的随机数序列，运行时间也只有250毫秒。

❶ 历史注解。对于表的排序类似在磁带上排序。随机访问是低效的；数据必须按顺序正向或反向地扫描。关于磁带排序方面的文献是很奇妙的，虽然已经很过时了，但却能提供有用的思想。*mergepairs*中使用的技术非常类似于20世纪60年代开发的摆动排序 (Knuth, 1973, 5.4.5)。

合并排序很少用于数组，因为它不能原地完成：需要两个数组。即便如此，早在1945年就已经被提出了；在输入中利用行程的思想也相当古老了 (Knuth, 1973, 5.2.4)。

练习 3.41 利用下面的函数来编写一个新版本的自顶向下合并排序，并测量它的速度。解释一下你的发现，如果可能的话，测量时把垃圾收集的时间也算进去。

```
fun alts ([], xs, ys) = (xs, ys)
  | alts ([x], xs, ys) = (x::xs, ys)
  | alts (x::y::l, xs, ys) = alts(l, x::xs, y::ys);
```

练习 3.42 和上面的练习一样，只是你要基于下面的函数编写新的排序函数：

```
fun takedrop ([], n, xs) = (xs, [])
  | takedrop (x::l, n, xs) =
    if n>0 then takedrop(l, n-1, x::xs)
    else (xs, x::l);
```

练习 3.43 为什么是调用*sorting*(*xs*, [[]], 0)，而不是调用*sorting*(*xs*, [], 0)？

练习 3.44 书写一个新版本的*samsort*，同时利用升序和降序行程。

多项式算术

计算机是为了进行数值运算而发明的。它们很善于用图解的方法描绘数据。但是有时没有什么可以比一个符号公式提供的信息更多了。公式 $E = mc^2$ 的图只是简单的一条直线。

计算机代数 (computer algebra) 所关心的是关于符号数学的自动化，科学家和工程师都要用到符号数学。像MACSYMA和REDUCE这样的系统可以完成涉及到微分、积分、幂级数扩展等惊人的任务。最基本的符号运算是进行多项式算术。即使是这个，也很难高效地完成。我们将把自己限制在最简单的情形——一元多项式上。一元多项式只有一个变元*x*；它们可以

非常直接地进行加法和乘法运算:

$$(x+1) + (x^2 - 2) = x^2 + x - 1$$

$$(x+1) \times (x^2 - 2) = x^3 + x^2 - 2x - 2$$

在为一元多项式开发算术包的时候, 涉及到了数据表示这个一般性的问题。我们要实现加法和乘法, 会用到前面章节的排序思想。我们还会遇到另一个满足2.22节ARITH签名的算术结构。最后, 要考虑如何找到最大公因式, 即使在一元的情况下, 这也是一个挑战。

这个代码速度很快, 能在2秒内计算 $(x^3 + 1)^{1000}$, 表现了该工具的强大能力。

3.22 表示抽象数据

在3.15节, 我们考虑过有限集合的操作, 例如并集和子集。ML并不提供有限集合作为一种数据结构; 我们是通过没有重复元素的表来表示它的。虽然有限集合可能看上去很简单, 但是它却展示了数据表示中涉及到的大部分问题。

一个抽象对象集, 比如有限集合, 是通过一个具体对象的集合, 比如某些表, 来表示的。每个抽象对象都至少可以表示为一个具体对象。有可能不止一个: 回忆一下 $\{3, 4\}$ 可以被表示为 $[3, 4]$ 和 $[4, 3]$ 。而有些具体对象, 比如 $[3, 3]$, 根本表示不了抽象对象。

抽象数据上的操作是根据其表示方法来定义的。例如, 只要对于分别表示集合A和A'的所有表l和l'来说, $\text{union}(l, l')$ 都表示A和A'的并集, 那么就可以说ML函数union实现了抽象函数“并集”。再如, 只要对于分别表示集合A和A'的所有表l和l'来说, 当且仅当A是A'的子集时, $l \text{ subs } l'$ 都为真, 那么就可以说ML的谓词subs (一个中缀操作符) 实现了抽象关系“属于”。相等关系也是类似看待的, 我们并没有要求相等的集合有相等的具体表示。

这些问题在我们每天使用计算机的时候都会出现, 计算机最终是用零和一来表示所有的数据的。一些更深入的问题只能在这里提一下, 例如计算机用浮点数来表示实数, 然而大部分的实数都无法这样表示, 实数运算只能近似地实现。

115

3.23 多项式的表示

让我们来考虑一下如何表示形如下式的一元多项式

$$a_n x^n + \dots + a_0 x^0$$

由于只有一个变元, 它的名字就没必要存储了。系数 a_n, \dots, a_0 有可能是实数, 但是实数运算在计算机里面又是近似的。我们应该把系数用有理数来表示, 也就是没有公约数的整数序偶 (参见练习2.25)。不过这只是些枝节问题, 再说它又需要任意精度的整数, 某些ML系统并不支持。因此, 在不是很影响严格性的情况下, 权宜的办法就是用ML的实数来表示多项式系数。

我们也许可以将多项式表示成系数的列表, $[a_n, \dots, a_0]$ 。但为了看出这个办法不合适, 不妨考虑一下多项式 $x^{100} + 1$, 以及它的平方! 在一个典型的多项式中许多系数都是零。我们需要的是一个稀疏 (sparse) 的表示方法, 而不是早先用来表示矩阵的密集 (dense) 方法。

让我们把多项式表示为代表每一个非零系数 a_k 的, 形如 (k, a_k) 的序偶表。序偶在表中应该按k的降序排列; 这样便于合并指数相同的项。例如, $[(2, 1.0), (0, 2.0)]$ 表示了多项式 $x^2 - 2$ 。

这个表示要比有限集合的表示好, 因为每一个抽象的多项式只有唯一的具体表示。两个多项式是相等的当且仅当表示它们的表是相等的。但是并不是所有 (整数, 实数) 的序偶表

都能表示多项式。

我们应该将多项式声明为抽象类型，隐藏下面的表。不过现在，还是让我们先把多项式操作按照签名`ARITH`包装到一个结构中去。

```
structure Poly =
  struct
    type t = (int*real) list;
    val zero = [];
    fun sum ...
    fun diff ...
    fun prod ...
    fun quo ...
  end;
```

116 这里，`t`是表示多项式的类型，`zero`是表示零多项式的空表。其他的组件将在下面介绍。

3.24 多项式加法和乘法

计算两个多项式的和就像合并对应的表，如 $(x^3 - x) + (2x^2 + 1) = x^3 + 2x^2 - x + 1$ 。不过，相似的要合并，且零系数项要消除，比如 $(x^4 - x + 3) + (x - 5) = x^4 - 2$ 。ML函数的定义是根据我们笔算的方法得来的：

```
fun sum ([], us) = us : t
  | sum (ts, []) = ts
  | sum ((m,a)::ts, (n,b)::us) =
    if m>n then (m,a) :: sum (ts, (n,b)::us)
    else if n>m then (n,b) :: sum (m, (m,a)::us)
    else (* m=n *)
      if Real.==(a+b,0.0) then sum (ts, us)
      else (m, a+b) :: sum (ts, us);
```

两个多项式的积是根据分配率来计算的。其中一个多项式的每一项都去乘以另一个多项式，然后将结果加起来。

$$\begin{aligned}(x^2 + 2x - 3) \times (2x - 1) &= x^2(2x - 1) + 2x(2x - 1) - 3(2x - 1) \\ &= (2x^3 - x^2) + (4x^2 - 2x) + (-6x + 3) \\ &= 2x^3 + 3x^2 - 8x + 3\end{aligned}$$

为了实现这个方法，首先需要有一个函数来将一项和一个多项式乘起来：

```
fun termprod ((m,a), []) = [] : t
  | termprod ((m,a), (n,b)::ts) =
    (m+n, a*b) :: termprod ((m,a), ts);
```

朴素 (naïve) 的乘法算法完全按照上面的例子：

```
fun nprod ([], us) = []
  | nprod ((m,a)::ts, us) = sum (termprod ((m,a), us),
    nprod (ts, us));
```

更快的乘法。实验表明`nprod`对于大型的多项式来说是太慢了。它需要超过两秒钟以及多次的垃圾收集才能完成 $(x^3 + 1)^{400}$ 的平方计算。（如此大型的多项式在计算机代数中很常见。）原因是`sum`合并的都是长度差得很远的表。如果`ts`和`us`各有100项，那么`termprod((m, a), us)`也只有

100项, 而 $nprod(ts, us)$ 却有可能多达10 000项。它们的和最多有10 100项, 只增长了1%。

合并排序启发了一个更快的算法。把其中一个多项式分成两半, 分别递归地计算它们和另一个多项式的积, 然后将两个积加起来。尽管这形成了和之前一样多的加法, 但是它们是平衡的: 我们计算 $(p_1 + p_2) + (p_3 + p_4)$, 而不是 $p_1 + (p_2 + (p_3 + p_4))$ 。平均来讲, 被加数都较小, 而且每加一次都会使结果长度加倍。

```
fun prod ([], us)      = []
  | prod ([ (m,a) ], us) = termprod ((m,a), us)
  | prod (ts, us)      =
    let val k = length ts div 2
    in sum (prod (List.take(ts,k), us),
            prod (List.drop(ts,k), us))
    end;
```

这个在计算 $(x^3 + 1)^{400}$ 的平方时是 $nprod$ 速度的三倍, 而且多项式越大, 速度越快。

运行一些例子。虽然我们尚未完全定义结构 $Poly$, 不妨先假设它已经定义好了, 并包括一个将多项式显示成字符串的函数 $show$ 。我们来计算这部分一开始的那两个多项式 $x + 1$ 和 $x^2 - 2$ 的和与积:

```
val p1 = [(1,1.0), (0,1.0)]
and p2 = [(2,1.0), (0,-2.0)];

Poly.show (Poly.sum (p1, p2));
> "x^2 + x - 1.0" : string
Poly.show (Poly.prod (p1, p2));
> "x^3 + x^2 - 2.0x - 2.0" : string
```

结构 $Poly$ 也提供指数操作。函数 $power$ 是像2.14节里那样定义的。为了看一个大一点的例子, 让我们计算一下 $(x - 1)^{10}$:

```
val xminus1 = [(1,1.0), (0,-1.0)];

Poly.show (Poly.power (xminus1, 10));
> "x^10 - 10.0x^9 + 45.0x^8 - 120.0x^7 + 210.0x^6
> - 252.0x^5 + 210.0x^4 - 120.0x^3 + 45.0x^2
> - 10.0x + 1.0" : string
```

下面是 $(x^2 - 2)^{150}$ 的头几项:

```
Poly.show (Poly.power (p2, 150));
> "x^300 - 300.0x^298 + 44700.0x^296
> - 4410400.0x^294 + 324164400.0x^292..." : string
```

练习 3.45 编写 $diff$, 计算两个多项式的差。利用 $termprod$ 来写只需一行代码, 不过效率不高。

练习 3.46 编写 $show$, 这个函数要产生像上面正文中那样的输出。(函数 $Real.toString$ 和 $Int.toString$ 将数转换成字符串。)

练习 3.47 给出一个具说服性的讨论, 说明 sum 和 $prod$ 都是符合多项式的表示方式的。

练习 3.48 我们所说的 sum 正确地计算了两个多项式的和, 这到底是什么含义? 你会怎样证明它?

117

118

3.25 最大公因式

很多应用都涉及到有理函数 (rational function): 多项式的分式。为了效率, 需要分式的分子和分母没有公因式。因此, 我们需要一个函数来计算两个多项式的最大公因式 (GCD)。这要求有函数来计算多项式的商式和余式。

多项式除法。多项式除法的算法类似于普通的长除法。实际上它要更容易些, 因为它不需要猜测商数。每一步都会消去被除式的首项, 这是通过用除式的首项去除被除式的首项来完成的。这样做的每一步都会产生一项商式。让我们用 $2x^2 + x - 3$ 除以 $x - 1$:

$$\begin{array}{r} 2x + 3 \\ x - 1 \overline{) 2x^2 + x - 3} \\ \underline{2x^2 - 2x} \\ 3x - 3 \\ \underline{3x - 3} \\ 0 \end{array}$$

这里, 余式是零。一般情况下, 余式是一个多项式, 它的首项指数 (称为它的次数) 要比除式的次数小。让我们将 $x^3 + 2x^2 - 3x + 1$ 除以 $x^2 + x - 2$, 得到余式 $-2x + 3$:

$$\begin{array}{r} x + 1 \\ x^2 + x - 2 \overline{) x^3 + 2x^2 - 3x + 1} \\ \underline{x^3 + x^2 - 2x} \\ x^2 - x + 1 \\ \underline{x^2 + x - 2} \\ -2x + 3 \end{array}$$

119 如果除式的首项系数不是1的话, 那么很可能会出现分数。这会令计算更加缓慢, 但它却不会使基本算法复杂化。函数 *quorem* 直接实现了刚才所示的方法, 它返回一个序偶 (商式, 余式)。其中商式的形成是倒着的。

```
fun quorem (ts, (n,b)::us) =
  let fun dividing ([], qs) = (rev qs, [])
      | dividing ((m,a)::ts, qs) =
          if m < n then (rev qs, (m,a)::ts)
          else dividing (sum (ts, termprod ((m-n, ~a/b), us)),
                        (m-n, a/b) :: qs)
  in dividing (ts, []) end;
```

因为 $(x - 1) \times (x + 1) = x^2 - 1$, 所以 $x^2 - 2$ 除以 $x - 1$ 得到商式 $x + 1$ 和余式 -1 。假设 *Poly* 包含 *quorem*, 以及可以显示一对多项式的函数 *showpair*:

```
Poly.quorem (p2, xminus1);
> [(1, 1.0), (0, 1.0)], [(0, ~1.0)]
> : (int * real) list * (int * real) list
Poly.showpair it;
> "x + 1.0,      - 1.0" : string
```

让我们运行上面显示的第二个除法的例子。

```
Poly.showpair
(Poly.quorem ((3,1.0), (2,2.0), (1,~3.0), (0,1.0)),
```

```

      [(2,1.0), (1,1.0), (0,~2.0)]);
> "x + 1.0,      - 2.0x + 3.0" : string

```

利用`quorem`我们可以很简单地定义商函数`quo`。加上练习3.45的`diff`函数，结构`Poly`就有了满足签名`ARITH`的所有组件。我们的算术结构现在包括了复数、二进制数、矩阵和多项式！但是，再次提醒一下，这些结构有很多重要的不同点，每个都有自己的附加组件，这些组件对于其他结构是没意义的。虽然可以做到让`Poly`满足签名`ARITH`，但是通常我们都会充分利用其中附加组件的长处。

对于多项式的欧几里得算法。现在可以利用欧几里得算法来计算GCD了。还记得吗？#2是用来提取序偶第二个分量的函数，这里用来提取多项式除法所产生的余式。

```

fun gcd ([], us) = us
  | gcd (ts, us) = gcd (#2(quorem (us,ts)), ts);

```

假设`Poly`包括了`gcd`，并将其作为一个组件，我们可以用它来试验几个例子。 $x^8 - 1$ 和 $x^3 - 1$ 的GCD是 $x - 1$ ：

```

Poly.show (Poly.gcd ([ (8,1.0), (0,~1.0) ],
                      [ (3,1.0), (0,~1.0) ]));
> "x - 1.0" : string

```

而 $x^2 + 2x + 1$ 和 $x^2 - 1$ 的GCD是 $-2x - 2$ 吗？

```

Poly.show (Poly.gcd ([ (2,1.0), (1,2.0), (0,1.0) ],
                      [ (2,1.0), (0,~1.0) ]));
> " - 2.0x - 2.0" : string

```

这个GCD应该是 $x + 1$ 。就这个具体的困难来说，虽然可以通过将所有系数除以首项系数来解决，但是还存在其他问题。这些问题令我们想到必须使用有理数（分数）算术：见下面的警告。那样的话，计算GCD通常都需要操作非常大的整数，即使是对于仅仅具有一位数系数的初始多项式。算法还需要做很多深入的提炼，通常必须要用到模运算。

⚠ 小心舍入误差。在GCD里，浮点数（实数类型）的使用显得非常糟糕，因为它的第一行需要检测余式是否为零。除法中的舍入误差会导致余式具有非常小但又非零的系数，会因此而错过公因式。对于其他应用，多项式算术都表现得很出色，因为此时舍入误差是可预测的。（我要感谢James Davenport关于这方面的分析。）

📖 进一步的阅读。Davenport等（1993）做了一个出色的关于计算机代数的介绍。其中第2章讲到了数据表示。例如，我们知道，多元多项式可以表示为一元多项式，其中多项式的系数是关于其他变元的一元多项式。这样一来， $y^2 + xy$ 是关于 y 的一元多项式，它的两个系数是分别是1和多项式 x ；我们也可以交换 x 和 y 的角色。那本书的第4章还讲到了高效计算GCD所涉及的令人生畏的复杂技术。

要点小结

- 表是从空表（`nil`或`[]`）开始建立的，利用`::`（“cons”）将元素连接在前面。
- 重要的库函数包括`length`、`take`、`drop`、中缀操作符`@`（连接）和`rev`（翻转）。

- 通过将它的结果在另一个参数中积累，递归函数可以避免低效的表连接操作。
- 相等多态性允许函数对它的参数进行相等测试。有关的例子有：测试元素和表的成员关系的`mem`，以及搜索序偶表的函数`assoc`。
- 常见的排序算法可以用表来实现，并具有出乎意料的效率。
- 表可以用来表示二进制数、矩阵、图、多项式等，这使得操作和运算的表达变得简洁。

121
}
122

第4章 树和具体数据

具体数据 (concrete data) 是由一些构造组成的, 这些构造可以被检视、分解或合并成更大的构造。表就是具体数据的一个例子。我们可以测试表是否为空, 也可以将非空表分解成表头和表尾。新的元素也可以加入到表中。这一章将介绍几种其他形式的具体数据, 包括树和逻辑命题。

ML的数据类型 (datatype) 声明可以定义一个新的类型以及它的构造子 (constructor)。在表达式中, 构造子创建数据类型的值; 在模式中, 构造子则描述了如何分解这样的值。数据类型可以表示包含多个子类的数据类, 就像是Pascal中的变体记录, 但没有那样复杂和不安全。递归的数据类型通常用来表示树。作用在数据类型上的函数是通过模式匹配来声明的。

特殊的数据类型 *exn* 是异常 (exception) 类型, 代表发生了错误情况。错误是可以发出信号以及被捕捉的。异常处理也是通过模式匹配来测试特定的错误的。

本章提要

本章讲述了数据类型、模式匹配、异常处理和树。包括以下几节:

- 数据类型声明。通过例子来说明数据类型、构造子和模式匹配。为了表示国王和他的臣民, 类型 *person* (人) 是由四类个体组成的, 并且每个个体都有各自关联的适合信息。
- 异常。这表示了一类错误的值。可以为每个可能的错误定义异常。抛出异常代表发出错误信号; 处理异常则代表了进行一个替代计算。
- 树。树是一种分支结构。二叉树是表的一般化, 它有着非常多的应用。
- 以树为基础的数据结构。字典、弹性数组和优先队列很容易用二叉数来实现。更新操作建立了新的结构, 不过会尽量减少复制。
- 重言式检测器。这是最基本的定理证明的例子。其中声明了关于命题 (布尔表达式) 的数据类型。函数将命题转换成合取范式并进行重言式测试。

123

数据类型声明

一个由不同成分构成的类包含了多个不同的子类。圆、三角形和正方形都是几何形状, 但种类却不同。三角形可以用三个点来表示, 正方形用四个点来表示, 而圆要用圆心和半径来表示。

我们来个难一点的问题, 考虑将不列颠王国的居民按他们的级别来分类。这包括了国王 (King)、贵族 (Peer)、骑士 (Knight) 和农民 (Peasant)。对于每一种人都要记录相应的信息:

- 国王就是国王, 没什么需要多说的。
- 贵族拥有爵位、封地和第几代。
- 骑士和农民都有名字。

在弱类型语言中, 可以直接表达这些子类。我们只要小心区分骑士和农民; 其他的自然有所

区别。在ML里可以这样试试

```
"King"
("Earl", "Carlisle", 7)      ("Duke", "Norfolk", 9)
("Knight", "Gawain")         ("Knight", "Galahad")
("Peasant", "Jack Cade")     ("Peasant", "Wat Tyler")
```

不幸的是，它们并不具有相同的类型！基于这种表示，没有任何一个ML函数可以同时处理国王和农民。

4.1 国王和他的臣民

一个包含有国王、贵族、骑士和农民的ML类型可以通过数据类型（datatype）声明来创建：

```
datatype person = King
                | Peer of string*string*int
                | Knight of string
                | Peasant of string;
> datatype person
> con King      : person
> con Peer      : string * string * int -> person
> con Knight    : string -> person
> con Peasant   : string -> person
```

上面声明了五样东西，分别叫作类型`person`和它的四个构造子（constructor）`King`、`Peer`、`Knight`和`Peasant`。

类型`person`包含且仅包含了由它的构造子所构造的值。注意，`King`具有类型`person`，而其他的构造子则如同函数那样返回属于此类型的东西。这样，下面的值都具有类型`person`：

```
King
Peer("Earl", "Carlisle", 7)  Peer("Duke", "Norfolk", 9)
Knight "Gawain"              Knight "Galahad"
Peasant "Jack Cade"          Peasant "Wat Tyler"
```

进一步来说，这些值都是不同的。没有人可以既是骑士又是农民；也没有贵族可以有两个不同的爵位。

类型`person`的值就像其他ML的值那样，可以是函数的参数和返回值，并且可以嵌在其他的数据结构中，比如表：

```
val persons = [King, Peasant "Jack Cade", Knight "Gawain"];
> val persons = [King, Peasant "Jack Cade",
>               Knight "Gawain"] : person list
```

由于每个`person`都是一个单一的构造，所以它可以被分解。作用在数据类型之上的函数可以通过包含构造子的模式来声明。就像处理表那样，可以分为几种情形。对一个人的称呼是根据他的级别来确定的，在构造称呼的时候使用了字符串连接操作（`^`）：

```
fun title King      = "His Majesty the King"
  | title (Peer(deg, terr, _)) = "The " ^ deg ^ " of " ^ terr
  | title (Knight name)      = "Sir " ^ name
  | title (Peasant name)     = name;
> val title = fn : person -> string
```

每个情形都由一个模式以及这个模式自有的变量来确定。在`Knight`和`Peasant`这两个情形中都

涉及变量名`name`，但是它们分属不同的作用域。

```
title(Peer("Earl", "Carlisle", 7));
> "The Earl of Carlisle" : string
title(Knight "Galahad");
> "Sir Galahad" : string
```

必要时，模式可以非常复杂，将元组、表构造子和数据类型构造子组合在一起。函数`sirs`返回`persons`表中所有骑士的名字：

125

```
fun sirs [] = []
  | sirs ((Knight s) :: ps) = s :: (sirs ps)
  | sirs (p :: ps) = sirs ps;
> val sirs = fn : person list -> string list
sirs persons;
> ["Gawain"] : string list
```

函数中的分情处理是按照情形的顺序来进行的。第三个情形（具有模式`p::ps`）在`p`是`Knight`的时候并不会被考虑，因此，不能将它从上文中去掉。有些人倾向于使各个情形互不相交，以助于数学上的论证。但是这样一来，就需要将情形`p::ps`替换成分开的`King`、`Peer`和`Peasant`，使程序变得更长、更慢和更不易读。`sirs`的第三个情形作为一个条件等式是非常合理的，它对于所有不是形如`Knight`的`p`都成立。

在两个人进行等级比较的时候，情形的顺序就更加重要了。与其测试所有的16种情形，我们不如仅仅测试那些返回为`true`的情形，而对于剩下的情形返回`false`，这样的话，总共只有7种情形。注意，通配符在模式中的使用是很频繁的。

```
fun superior (King, Peer _) = true
  | superior (King, Knight _) = true
  | superior (King, Peasant _) = true
  | superior (Peer _, Knight _) = true
  | superior (Peer _, Peasant _) = true
  | superior (Knight _, Peasant _) = true
  | superior _ = false;
> val superior = fn : person * person -> bool
```

练习 4.1 书写一个ML函数将人映射到整数，将国王对应为4，贵族对应为3，骑士对应成2以及农民对应成1。书写一个与`superior`等价的函数，通过对相应映射结果的比较来完成等级比较。

练习 4.2 修改类型`person`，加入构造子`Esquire`（乡绅），他的参数是名字和所在的村庄（都使用字符串表示）。这个构造子的类型是什么？修改函数`title`以产生类似

```
"John Smith, Esq., of Bottisham"
```

的称呼。修改函数`superior`将`Esquire`排在`Knight`和`Peasant`之间。

练习 4.3 声明一个关于几何图形的数据类型，例如三角形、长方形、直线和圆。声明一个计算图形面积的函数。

126

4.2 枚举类型

用字符串来表示贵族的爵位可能是不可取的。这样无法防止假的爵位，比如“`butcher`”

(屠夫)和“madman”(疯子)。对仅有的五个有效爵位；我们将它们声明为新数据类型的构造子：

```
datatype degree = Duke | Marquis | Earl | Viscount | Baron;
(* 公爵、侯爵、伯爵、子爵和男爵 *)
```

现在需要重新声明类型`person`，使得构造子`Peer`具有类型

$$\text{degree} \times \text{string} \times \text{int} \rightarrow \text{person}$$

对于类型`degree`的函数也是通过分情来处理。我们看看上流社会女性的称呼是怎样的？

```
fun lady Duke    = "Duchess"
  | lady Marquis = "Marchioness"
  | lady Earl    = "Countess"
  | lady Viscount = "Viscountess"
  | lady Baron   = "Baroness";
> val lady = fn : degree -> string
```

准确的称呼在法庭和社交专栏上是至关重要的，但在电子出版方面，这个例子的重要性就不那么明显了。

像`degree`这样由有限数目的常量组成的类型称为枚举类型 (enumeration type)。另外一个例子就是内置的`bool`类型，它是这样声明的

```
datatype bool = false | true;
```

函数`not`是通过分情来声明的

```
fun not true = false
  | not false = true;
```

标准库声明了枚举类型`order` (顺序)，如下：

```
datatype order = LESS | EQUAL | GREATER;
```


这涵盖了比较的三种可能结果。程序库的字符串、整数、实数、时间、日期等的结构都各自包括了一个比较函数`compare`，此函数将返回这三种结果中的一个：

```
String.compare ("York", "Lancaster");
> GREATER : order
```

我们对于返回布尔值的关系操作可能更为熟悉。但是需要调用两次`<`来取得一次`String.compare`调用所能返回的信息。第一个Fortran版本就提供了返回三种结果的比较操作！

127

在计算机技术的不断演变中，它们一直维持原状……

 小心数据类型的重定义。每个数据类型声明都会创建一个与其他类型都不同的新类型。假设我们已经定义了类型`degree`和函数`lady`。现在，重复声明一次`degree`，这会声明全新的类型和构造子。试图对`lady(Duke)`求值会得到类型错误“期望类型`degree`，但遇到类型`degree`。”两个不同的类型现在都叫`degree`。这种令人恼火的情形可能出现在交互式修改程序的过程中。最彻底的补救方法是终止当前的ML会话，启动一个新的会话，然后重新调入程序。

练习 4.4 声明一个枚举类型，由6个不同的国家名组成。书写一个函数将每个国家的首都名字作为字符串返回。

练习 4.5 书写类型为 $bool \times bool \rightarrow bool$ 的函数，实现布尔的合取和析取。使用模式匹配，而不是`andalso`、`orelse`或者`if`。这需要显式地测试多少种情形？

4.3 多态数据类型

回想一下`list`是具有一个参数的类型操作符。[⊖]由此可知，`list`就不是一种类型，而 $(int) \text{ list}$ 和 $((string \times real) \text{ list})$ 却是类型。数据类型声明可以引入新的类型操作符。

“可选”类型。标准库定义了类型操作符`option`：

```
datatype 'a option = NONE | SOME of 'a;
> datatype 'a option
>   con NONE : 'a option
>   con SOME : 'a -> 'a option
```

类型操作符`option`有一个参数。类型 $\tau \text{ option}$ 包括了类型 τ 的一个副本，并增加了额外的一个值`NONE`。它可以用来向函数提供可选的参数，但是最明显的用途还是指示错误。例如，库函数`Real.fromString`将字符串参数转换成实数，但是它不能接受表示60 000的所有表达方式：

```
Real.fromString "6.0E5";
> SOME 60000.0 : real option
Real.fromString "full three score thousand";
> NONE : real option
```

可以使用`case`表达式来分情处理返回的结果，见下面4.4节。

不相交和类型。这是一个基本的操作符，构成两个类型的不相交和（disjoint sum）或联合（union）：

```
datatype ('a,'b) sum = In1 of 'a | In2 of 'b;
```

类型操作符`sum`有两个参数。它的构造子分别是

$$\begin{aligned} In1 : \alpha &\rightarrow (\alpha, \beta) \text{ sum} \\ In2 : \beta &\rightarrow (\alpha, \beta) \text{ sum} \end{aligned}$$

类型 $(\sigma, \tau) \text{ sum}$ 是类型 σ 和 τ 的不相交和。如果 x 属于类型 σ ，它的值形如 $In1(x)$ ，或者如果 y 属于类型 τ ，它的值则形如 $In2(y)$ 。这个类型含有 σ 和 τ 的副本。可以观察到， $In1$ 和 $In2$ 可以被看作是区分 σ 和 τ 的标签。

不相交和允许在通常只能出现一个类型的场合里放上多个类型。表的元素必须具有相同的类型，如果这个元素类型是 $(string, person) \text{ sum}$ ，那么元素就可以是字符串或者是“人”，而类型 $(string, int) \text{ sum}$ 则包括了字符串和整数。

```
[In2(King), In1("Scotland")] : ((string, person) sum) list
[In1("tyrant"), In2(1040)]   : ((string, int) sum) list
```

不相交和的模式匹配可以测试出元素是存在于 $In1$ 中还是存在于 $In2$ 中的。函数`concat1`将表中所有 $In1$ 中的字符串连接起来：

⊖ 正确的名称是类型构造子（type constructor）（Milner等，1990）。我在这里避免使用这个名称是为了不造成它和数据类型的构造子之间的混淆。

```

fun concat1 [] = ""
  | concat1 ((In1 s)::l) = s ^ concat1 l
  | concat1 ((In2 _)::l) = concat1 l;
> val concat1 = fn : (string, 'a) sum list -> string
concat1 [ In1 "O!", In2 (1040,1057), In1 "Scotland" ];
> "O!Scotland" : string

```

表达式`In1 "Scotland"`已经出现在两种类型中了,也就是 $(string, int \times int)sum$ 和 $(string, person)sum$ 。这是可能的,因为这个表达式的类型是多态的:

```

In1 "Scotland";
> In1 "Scotland" : (string, 'a) sum

```

表达其他的数据类型。不相交和可以表达所有的非递归数据类型。类型`person`可以表示为

129

$$((unit, string \times string \times int)sum, (string, string)sum)sum$$

而原来的构造子映射为

$$\begin{aligned}
 King &= In1(In1()) \\
 Peer(d, t, n) &= In1(In2(d, t, n)) \\
 Knight(s) &= In2(In1(s)) \\
 Peasant(s) &= In2(In2(s))
 \end{aligned}$$

这些既可以作为表达式也可以作为模式。不用说,类型`person`更好看一些。注意观察唯一的国王是怎样通过只有一个值`()`的类型`unit`来表示的。

❶ 空间的需求。数据类型需要令人吃惊的大量空间,至少现在的编译器是这样的。典型的值需要4字节的标记(来标识构造子),相关元组的每个分量各需要4字节(指针)。垃圾收集器需要一个头,又占去了4字节。结果对于`Knight`和`Peasant`总共各需要12字节,而`Peer`则需要20个字节。这里会直接将整数存放在`Peer`里面,而字符串则需要作为单独的对象另外存放。

枚举类型的内部值需要的空间不会比整数更多,尤其是在那些允许无限精度整数的ML系统中。表元素一般占用8~12个字节。如果使用一个过时的垃圾收集器,一个对象所占用的空间还可能随着对象的生存时间而变化!

也有可能做出优化。如果一个数据类型只有一个构造子,那么就没有必要存储标记。如果只有一个构造子不是常量,那么这个构造子可能也不需要标记;这对于表是可以的,但对于可选类型就不行,因为`SOME`的参数可以是任何东西。[⊖]和Lisp比较,运行时没有类型是可以节省一些空间的。Appel (1992)讨论了这方面的问题。随着运行时系统的改进,我们可以期望对于空间的需求还会适当减小。

练习 4.6 如上定义的`King`、`Peer`、`Knight`和`Peasant`都是什么类型?

练习 4.7 展示类型 $(\sigma, \tau)sum$ 的值和类型 $(\sigma list) \times (\tau list)$ 的某些值——形如 $([x], [])$ 或 $([], [y])$, 的对应关系。

[⊖] 这里的意思大概是指针值和小整数是可以区分的,如果构造子的参数包含有对于其他对象的引用(指针),那么这个构造子通过指针值就可以和其他常量构造子区分开来。——译者注

4.4 通过val、as、case进行模式匹配

模式 (pattern) 是一个只包含变量、构造子和通配符的表达式。构造子包括

- 数、字符和字符串常量
- 序偶、元组和记录结构
- 表和数据类型的构造子

130

在模式中，所有不是构造子的名字都是变量。任何它们在模式之外可能拥有的意思都无效了。模式中的变量必须彼此不同。这些条件保证了值可以有效地和模式进行匹配，并且以唯一的方式通过分析绑定到变量上去。

构造子必须绝对和变量区分开来。在本书中，构造子以大写字母开头，而大多数变量则以小写字母开头。[⊖]但是，标准的构造子`nil`、`true`和`false`仍旧以小写字母开头。构造子的名字可以是符号的，也可以是中缀的，例如表的构造子`::`。标准库喜欢将构造子命名为全大写字母，就像`NONE`。



模式匹配中的错误。模式中的拼写错误是很难发现的。下面版本的函数`title`含有几个错误，试着把它们找出来，然后再往下读：

```
fun title Kong           = "His Majesty the King"
  | title (Peer(deg,terr,_)) = "The " ^ deg ^ " of " ^ terr
  | title (Knightname)     = "Sir " ^ name
  | title Peasant name     = name;
```

第一个错误是把构造子`King`错拼成`Kong`了。这就变成了一个变量，而且匹配所有的值，它阻止了继续对其他情形的考虑。当函数有冗余的情形时，ML编译器会给出警告，对这个警告必须留意！

第二个错误是`Knightname`：漏掉一个空格再次将构造子变为了变量。由于这个错误使得`name`成了未定义的变量，因此编译器会报错。

第三个错误是漏掉了`Peasant name`外面的括号，所得出的错误信息有可能让人莫名其妙。

拼写错误的构造子函数（带参数的构造子）会被立即检测出来，因为

```
fun f (g x) = ...
```

只有在`g`是构造子的情况下才是允许的。其他的拼写错误可能不会引起任何警告。漏掉通配符前面的空格，比如`Peer_`，这种错误是特别隐蔽的。

练习 4.8 在`superior`中犯什么样的简单错误会只改变函数的行为而又不会引起冗余情形？

值声明中的模式。声明

```
val P = E
```

定义了模式`P`中的变量，并赋予它们表达式`E`中相应的值。我们曾经在第2章使用这一方法来选择元组中的分量：

131

⊖ Haskell语言强制这种约定。

```
val (xc,yc) = scalevec(4.0, a);
> val xc = 6.0 : real
> val yc = 27.2 : real
```

我们也可以书写

```
val [x,y,z] = upto(1,3);
> val x = 1 : int
> val y = 2 : int
> val z = 3 : int
```

如果表达式的值与模式不匹配的话，声明就会失败（抛出一个异常）。当模式是元组的时候，类型检测可以避免这样的异常。

下面的声明是有效的：表达式的值与它们的模式匹配，却没有声明任何值。

```
val King = King;
val [1,2,3] = upto(1,3);
```

构造子的名字不能用`val`声明来改变它的用途。在类型`person`的作用域内，名字`King`、`Peer`、`Knight`和`Peasant`都被保留作构造子。像下面这样的声明会被认为是试图做模式匹配，并会被拒绝而得到类型错误的信息：

```
val King = "Henry V";
val Peer = 925;
```

多层模式（layered pattern）。模式中的变量还可以是下面的形式

Id as P

如果整个模式（*P*是其中的一部分）匹配的话，那么和*P*匹配的值也会被绑定到标识符*Id*上。这个值既可以通过模式来看，也可以通过整体来看。函数`nextrun`（出自3.21节）可以这样编写

```
fun nextrun(run, []) = ...
  | nextrun(run as r::_, x::xs) =
      if x < r then (rev run, x::xs)
      else nextrun(x::run, xs);
```

132

这里`run`和`r::_`是同一个表。现在可以通过`r`来引用表头而不需要`hd run`了。这种写法是否比前一个版本更易读是存在争议的。

分情（`case`）表达式。这是模式匹配的另一个载体，它的形式是

`case E of P1 => E1 | ... | Pn => En`

*E*的值顺序地和模式 P_1, \dots, P_n 进行匹配；如果 P_i 是第一个匹配的模式，那么 E_i 的值就是整个表达式的结果。因此，如果根据这些分情定义一个函数，并把该函数应用在*E*上面，那么这个函数表达式将和分情表达式等价。通常的分情表达式都是测试几种显式的值，然后以一个通配的情形结束：

```
case p-q of
  0 => "zero"
  | 1 => "one"
  | 2 => "two"
  | n => if n < 10 then "lots" else "lots and lots"
```

函数`merge`（同样出自3.21节）可以重新利用`case`编码，先测试第一个参数然后再测试第二

个参数:

```
fun merge(xlist, ylist) : real list =
  case xlist of
    [] => ylist
  | x::xs => (case ylist of
               [] => xlist
             | y::ys => if x<=y then x::merge(xs, ylist)
                        else y::merge(xlist, ys));
```

在递归调用中, *xlist*和*x::xs*表示同一个表, 这个效果也可以通过模式*xlist as x::xs*得到。

△ *case*的作用域。没有专门的符号来结束一个分情语句, 因此, 如果你不能肯定语句没有歧义, 则应用括号把整个语句括起来。下面, 虽然看起来程序员是要把第二行归于外面的分情语句, 但是它却是里面分情语句的一部分:

```
case x of 1 => case y of 0 => true | 1 => false
        | 2 => true;
```

下面的声明在语法上并没有歧义, 但是很多ML编译器都不能正确地进行分析。分情语句需要用括号括起来:

```
fun f [x] = case g x of 0 => true | 1 => false
              | f xs = true;
```

练习 4.9 利用分情语句来区分类型*person*的四个构造子, 以这种方式来表达函数*title*。

133

练习 4.10 叙述一种简单的方法将所有的分情语句从程序中去掉。解释一下为什么你的方法不会改变程序的意思。

异常

一个困难的问题可能需要多种方法来解决, 而每一种方法只可以成功地处理其中的部分情形。除了实际试一试某种方法是否成功以外, 可能没有更好的办法来选择方法。如果计算走进了死胡同, 那么这个方法就失败了, 或者得出这个问题是不可能解决的结论。某个证明方法可能是毫无进展的, 也可能把它的证明目标简化成了 $0 = 1$ 。某个数值的算法可能会遭遇到溢出或除数为零的错误。

这些结果可以表示为一个数据类型, 这个类型的值是*Success(s)* (成功), 其中*s*是一个解、*Failure* (失败) 和*Impossible* (不可能或无解)。处理多种返回结果是复杂的, 就像我们在3.17节的拓扑排序函数里看到的那样。返回成功和失败的信息的函数*cyclesort*要比*pathsort*复杂, 后者通过调用*hd []*来表达失败。(非常难看!)

ML通过异常(exception)来处理失败。在发现失败的地方抛出(raise)异常, 并在其他地方处理(handle), 有可能是在距离很远的地方。

4.5 异常初步

异常是错误值的数据类型, 为了减少显式的测试, 对异常进行了特别的处理。当异常被抛出时, 会被所有的ML函数传递, 直到某个异常处理器(exception handler)发现它。异常处理器基本上就是一个分情表达式, 它描述了对于每一种异常的返回值。

假设函数`methodA`和`methodB`用不同的方法来解题，并且`show`将解显示成字符串。使用数据类型和它的构造子`Success`、`Failure`和`Impossible`，可以通过嵌套的分情语句来尝试解题并显示其结果。如果`methodA`失败了，那么就试试`methodB`，若两个都失败了的话，则计算结束。对于所有的情况，结果都是同一个类型：`string`。

```
case methodA (problem) of
  Success s => show s
| Failure   => (case methodB (problem) of
                  Success s => show s
                  | Failure   => "Both methods failed"
                  | Impossible => "No solution exists")
| Impossible => "No solution exists"
```

134

现在来试试异常处理。声明异常`Failure`和`Impossible`来代替可能出现的结果的数据类型：

```
exception Failure;
exception Impossible;
```

函数`methodA`和`methodB`，以及任何它们所调用的函数，只要在上面异常声明的作用域内，都可以通过类似下面的代码发出错误信号

```
if ... then raise Impossible
else if ... then raise Failure
else (* 计算成功的结果 *)
```

对于`methodA`和`methodB`的调用尝试涉及到两个异常处理器：

```
show (methodA (problem)
      handle Failure => methodB (problem))
      handle Failure   => "Both methods failed"
      | Impossible => "No solution exists"
```

第一个处理器从`methodA`中捕获`Failure`，然后尝试`methodB`。第二个处理器从`methodB`中捕获`Failure`并从两个方法中捕获`Impossible`。如果`methodA`成功的话，函数`show`被给予`methodA`的结果，否则就是`methodB`的结果。

即使在这样简单的例子中，使用异常也能得到较短、较清晰和较快的程序。错误信息的传播并没有使程序变得混乱。

4.6 声明异常

异常的名字在Standard ML里面是内置类型`exn`的构造子。该数据类型有一个独特的性质：它的构造子集合可以被扩展。异常声明

```
exception Failure;
```

使得`Failure`成为类型`exn`的一个新构造子。

虽然`Failure`和`Impossible`都是常量，但是构造子也可以是函数：

```
exception Failedbecause of string;
exception Badvalue of int;
```

构造子`Failedbecause`（表示错误原因）具有类型`string → exn`，而`Badvalue`（表示无效值）具有类型`int → exn`。它们可以创建异常`Failedbecause(msg)`，`msg`是供显示用的错误信息，以及异常

Badvalue(*k*), 而整数*k*可以用来决定下面接着要尝试哪个方法。

异常可以通过`let`在局部声明, 甚至可以在一个递归函数的内部声明。这会造成不同的异常具有相同的名字以及其他的复杂情况。在可能的情况下, 尽量将异常声明在顶层。顶层的异常必须是单态的。[⊖]

类型*exn*的值可以存放在表里, 也可以作为函数的返回值, 等等, 就像其他类型的值一样。另外, 它们在`raise`和`handle`的操作中扮演了特殊的角色。

i 动态类型和*exn*。由于类型*exn*可以通过增加构造子来进行扩展, 所以它可以暗含任意类型的值。于是, 我们得到了一种弱形式的动态类型机制。这是ML的一个附带特性; CAML则是用更成熟的方式来处理动态 (Leroy和Mauny, 1993)。

例如, 假设我们希望提供一种统一的接口来将任意数据表示成字符串。所有的转换函数都可以具有类型*exn* → *string*。如果想将这个系统扩展到一个新的类型上, 比如说*Complex.t*, 我们为这个类型定义了新的异常, 并书写类型为*exn* → *string*的新转换函数:

```
exception ComplexToString of Complex.t;
fun convert_complex (ComplexToString z) = ...
```

只有当这个函数应用于构造子*ComplexToString*的时候, 才是有用的。一组类似的函数可以被存储在字典里面, 通过统一的键值, 例如字符串, 来标识。这样, 就有了面向对象程序设计的基本形式。

4.7 抛出异常

异常的抛出会建立一个异常包 (exception packet), 里面包括了类型*exn*的一个值。如果*Ex*是类型为*exn*的表达式, 并且计算*Ex*得到值*e*, 那么

```
raise Ex
```

则会计算得出一个包含值*e*的异常包。异常包并不是ML的值, 能识别它们的只有`raise`和`handle`操作。因此, 类型*exn*起到了协调异常包和ML值的作用。

在计算过程中, 异常包的传播遵循传值调用的原则。如果表达式*E*返回一个异常包, 那么对于任意函数*f*, 函数应用*f*(*E*)的结果也是这个异常包。因此, *f*(`raise Ex`)和`raise Ex`等价。另外, `raise`本身也传播异常, 所以

```
raise (Badvalue (raise Failure))
```

抛出异常*Failure*。

在ML中, 表达式是从左到右计算的。如果*E*₁返回异常包, 那么这也是序偶(*E*₁, *E*₂)的结果, 表达式*E*₂根本不进行求值。如果*E*₁返回一个正常值而*E*₂返回异常包, 那么该异常包就是序偶的值。当*E*₁和*E*₂抛出不同的异常时, 结果就和求值的顺序有关了。

求值的顺序在条件表达式中也可以察觉:

```
if E then E1 else E2
```

[⊖] 这个限制和命令式的多态性有关; 见8.3节。

如果 E 的计算结果是`true`,那么只有 E_1 会被求值。它的值,不管正常与否,都将成为条件表达式的值。类似地,如果 E 的计算结果是`false`,那么只有 E_2 会被求值。还有第三种可能性,如果测试 E 抛出一个异常,那么它就成为了条件表达式的结果。

最后,考虑`let`表达式

```
let val P = E1 in E2 end
```

如果 E_1 的计算结果是异常包的话,那么整个`let`表达式的结果也是。

异常包并不是通过测试来进行传播的。ML系统可以高效地跳转到正确的异常处理器,如果没有这样的异常处理器,运行就要终止了。

标准异常。失败的模式匹配可能会抛出内置的异常`Match`或者`Bind`。函数在被应用到和它的所有模式都不匹配的参数上时则抛出异常`Match`。当分情表达式没有匹配的模式时,也会抛出异常`Match`。ML遇到非穷尽模式时(不能概括该类型所有的值)会事先提出警告,表示有这种异常的可能。

由于很多函数都可能抛出`Match`,使得这个异常传递的信息有限。在编写函数时,可以让它通过显式地抛出恰当的异常来拒绝不正确的参数,最后一个情形可以用来捕捉任何不能匹配其他模式的值。有些程序员为每一个函数声明一个异常,但是太多的异常会导致混乱。标准库走的是中间路线,为一整类的错误声明异常。下面是一些例子。

- `Overflow`是针对结果超出范围的算术操作的。
- `Div`是针对除数为零的。
- `Domain`是针对涉及结构`Math`的函数错误的,比如对负数取平方根或对数。
- `Chr`是针对`chr(k)`的,当 k 是无效的字符编码时抛出`Chr`。
- `Subscript`是针对下标越界的。数组、字符串和表的操作可能抛出`Subscript`。
- `Size`是针对尝试建立大小为负数或过大的数组、字符串或表的。
- `Fail`是针对一些杂类错误的,它带有一个字符串参数来表示错误信息。

库结构`List`声明了异常`Empty`。函数`hd`和`tl`在应用到空表上面的时候会抛出这个异常:

```
exception Empty;
fun hd (x::_) = x
  | hd [] = raise Empty;
fun tl (_,xs) = xs
  | tl [] = raise Empty;
```

返回表的第 n 个元素(从0开始计)的库函数则不那么简单:

```
exception Subscript;
fun nth (x::_, 0) = x
  | nth (x::xs, n) = if n>0 then nth (xs,n-1)
                     else raise Subscript
  | nth _ = raise Subscript;
```

对`nth(l, n)`的计算在 $n < 0$ 或 l 里面没有第 n 个元素时会抛出异常`Subscript`。在后一种情况下,异常会在一系列对`nth`的递归调用中向上传播。

```
nth(explode "At the pit of Acheron", 5);
> #"e" : char
```

```
nth([1,2], 2);
> Exception: Subscript
```

当 E 的值不能匹配模式 P 的时候, 声明`val P = E`抛出异常`Bind`。这种风格通常不太好(不过请见图8-4)。如果存在值不能匹配模式的可能, 那么应该考虑替代的办法, 显式地使用分情表达式:

```
case E of P1 => ... | P2 => ...
```

4.8 处理异常

异常处理器测试表达式的结果是否是异常包。如果是, 那么异常包的内容, 一个类型为`exn`的值, 可以被分情处理。具有异常处理器的表达式在构造上与分情表达式很类似:

```
E handle P1 => E1 | ... | Pn => En
```

如果 E 返回正常值, 异常处理器简单地将它传出去。相反, 如果 E 返回一个异常包, 那么它的内容就要和那些模式进行匹配。如果 P_i 是第一个匹配的模式, 则 E_i 就是要返回的值, 这里 $i = 1, \dots, n$ 。

有一点和分情表达式截然不同。如果没有匹配的模式, 那么异常处理器会继续传播那个异常包, 而不是抛出异常`Match`。通常, 一个异常处理器并不会考虑所有可能的异常。

在3.7节中, 我们考虑过找零钱的问题。那时的贪婪式算法(函数`change`)不会将16表示为5和2的组合, 这是因为它总是取最大的硬币。而另一个函数, `allChange`, 则通过返回所有的解来处理这个问题。

利用异常, 可以很容易地编写回溯算法。我们声明异常`Change`, 并在两种情况下抛出: 一种是在非零余额时没有硬币可选, 另一种是余额为负数。我们总是尝试取最大的硬币, 并在出错时退回一步。此时, 异常处理器总是退回最近的一次选择, 递归调用保证了这一点。

```
exception Change;
fun backChange (coinvals, 0)      = []
  | backChange ([], amount)      = raise Change
  | backChange (c::coinvals, amount) =
    if amount < 0 then raise Change
    else c :: backChange(c::coinvals, amount-c)
    handle Change => backChange(coinvals, amount);
> val change = fn : int list * int -> int list
```

不像`allChange`, 上面的函数最多返回一个解。让我们通过重复3.7节的例子来比较一下两个函数:

```
backChange([10,2], 27);
> Exception: Change
backChange([5,2], 16);
> [5, 5, 2, 2, 2] : int list
backChange(gb.coins, 16);
> [10, 5, 1] : int list
```

分别是无解、两个解和25个解, 我们最多能得到其中的一个。类似的关于异常处理的例子会出现在本书稍后章节的语法分析与合一的问题中。惰性表(5.19节)是异常处理之外的另一个选择, 利用它, 可以按需要生成多个解。

▲ 异常处理的缺陷。就像其他形式的模式匹配一样，必须小心地编写异常处理器。千万不要把异常的名字拼错，它会被当作变量而匹配所有的异常。

小心地赋予异常处理器正确的作用域。在表达式

```
if E then E1 else E2 handle ...
```

中，异常处理器只会检测到 E_2 抛出的异常。将条件表达式用括号括起来，则会使整个表达式置于异常处理器的作用域内。类似地，在

```
case E of P1 => E1 | ... | Pn => En handle ...
```

里，异常处理器只会检测到 E_n 抛出的异常。

分情表达式中的异常处理器可能会出现语法上的歧义。如果漏掉了下面的括号，就会使分情表达式的第二行归属于异常处理器：

```
case f u of [x] => (g x handle _ => x)
           | xs => g u
```

4.9 对异常的异议

异常可以成为模式识别的一个笨拙的替代品，如同这个计算表长度的函数中所表示的那样：

```
fun len l = 1 + len(tl l) handle _ => 0;
> val len = fn : 'a list -> int
```

用▲表示异常包， $len[1]$ 的计算过程是这样的：

```
len[1] => 1 + len(tl[1]) handle _ => 0
      => 1 + len[] handle _ => 0
      => 1 + (1 + len(tl[])) handle _ => 0) handle _ => 0
      => 1 + (1 + len▲ handle _ => 0) handle _ => 0
      => 1 + (1 + ▲ handle _ => 0) handle _ => 0
      => 1 + (▲ handle _ => 0) handle _ => 0
      => 1 + 0 handle _ => 0
      => 1
```

这个计算要比用模式匹配定义的长度函数更复杂。在可能的情况下，尽量事先分情处理，而不是不分分说地利用异常处理来乱试一气。

大多数惰性求值的支持者都反对异常处理。异常使得理论更复杂了，而且有可能被误用，就像我们刚才所看到的。实际的矛盾更为深入。异常是根据传值调用的原则传播的，而惰性求值则是遵循传需调用的原则传播的。

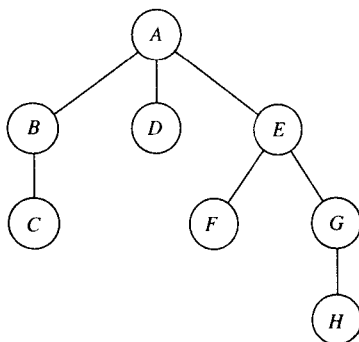
ML包括赋值命令和其他命令，而对命令式程序设计来说，异常可能是危险的。当执行可能在任意地方中断时，写出正确的程序是很困难的。限于程序的函数式部分，异常可以被理解为将值空间划分为普通值和异常包。虽然严格来讲，在程序设计语言中异常并不是必需的，而且还可能被误用，但是它却可以使程序更清晰、更高效。

练习 4.11 类型 exn 不允许进行ML的相等测试。这个限制合理吗？

练习 4.12 根据你的经验叙述一个适合运用异常处理的计算问题。书写一个ML程序的框架来解决这个问题。要包括异常的声明，以及讲述在什么地方抛出和处理异常。

树

树 (tree) 是一种分支结构，它由结点 (node) 组成，这些结点包含有通向子树的分支 (branch)。结点可以带有值，称为标签 (label)。尽管叫做树，但通常这个数据结构的树都是倒着画的：



标签为A的结点是树的根 (root)，而结点C、D、F和H (它们没有子树) 是树的叶子 (leaf)。

结点的类型决定了标签的类型，以及它可以有多少棵子树。而树的类型又决定了结点的类型。有两种类型的树是非常重要的。第一种树中有带标签的结点，每个结点有一个分支，最后以一个没有标签的叶子结束，这种树就是表。第二种树和表的不同之处在于有标签的结点都有两个分支而不是一个，这种树叫做二叉树 (binary tree)。

当函数式程序员使用表的时候，他们可以依赖成套的技术和函数库。而当使用树时，通常都要自己来安排了。这很遗憾，因为二叉树对于许多应用都是很理想的。下面的几节会将二叉树应用到查询、数组和优先队列上。我们要为二叉树开发一个多态函数库。

141

4.10 二叉树类型

二叉树具有分支结点，每个结点有一个标签和两棵子树。二叉树的叶子是没有标签的。要在ML里定义二叉树需要递归的数据类型声明：

```
datatype 'a tree = Lf
              | Br of 'a * 'a tree * 'a tree;
```

可以像非递归数据类型那样来理解递归数据类型。类型 $\tau \text{ tree}$ 是由所有可以通过 Lf 和 Br 构成的值组成的。至少存在一个 $\tau \text{ tree}$ ，即 Lf ；并且给定两棵树和一个类型为 τ 的值，我们可以构造另一棵树。因此， Lf 是递归的基本情形。

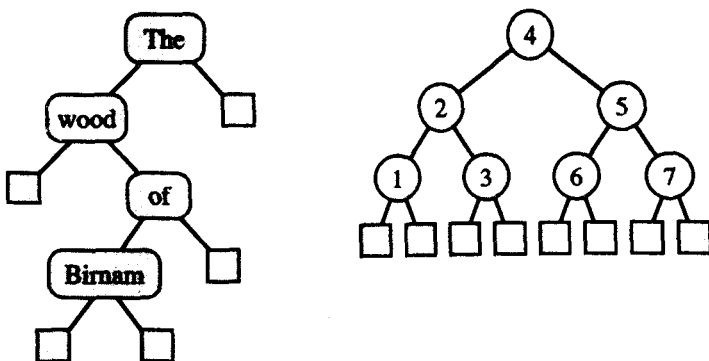
下面是一棵以字符串为标签的树：

```
val birnam =
  Br("The", Br("wood", Lf,
                Br("of", Br("Birnam", Lf, Lf),
                          Lf)),
    Lf);
> val birnam = Br ("The", ... , Lf) : string tree
```

下面是一些以整数为标签的树。注意树是怎样被组成更大的树的。

```
val tree2 = Br(2, Br(1,Lf,Lf), Br(3,Lf,Lf));
> val tree2 = Br (2, Br (1, Lf, Lf),
>               Br (3, Lf, Lf)) : int tree
val tree5 = Br(5, Br(6,Lf,Lf), Br(7,Lf,Lf));
> val tree5 = Br (5, Br (6, Lf, Lf),
>               Br (7, Lf, Lf)) : int tree
val tree4 = Br(4, tree2, tree5);
> val tree4 =
> Br (4, Br (2, Br (1, Lf, Lf),
>               Br (3, Lf, Lf)),
>      Br (5, Br (6, Lf, Lf),
>            Br (7, Lf, Lf))) : int tree
```

142 树birnam和tree4如下图所示:



叶子在上面用方块表示，不过以后就会被省略了。

树的操作表达为使用模式匹配的递归函数，多态函数size返回了树中的标签数：

```
fun size Lf = 0
  | size (Br(v,t1,t2)) = 1 + size t1 + size t2;
> val size = fn : 'a tree -> int
size birnam;
> 4 : int
size tree4;
> 7 : int
```

对于树的大小的另一个量度标准是它的深度depth：从根到叶子的最长路径。

```
fun depth Lf = 0
  | depth (Br(v,t1,t2)) = 1 + Int.max(depth t1, depth t2);
> val depth = fn : 'a tree -> int
depth birnam;
> 4 : int
depth tree4;
> 3 : int
```

可以看到birnam是相当深的，而tree4则是尽可能地浅。如果t是一棵二叉树，则有

$$\text{size}(t) \leq 2^{\text{depth}(t)} - 1$$

如果t满足 $\text{size}(t) = 2^{\text{depth}(t)} - 1$ ，那么它就是一棵完全二叉树 (complete binary tree)。例如，tree4就是一棵深度为3的完全二叉树。

通俗地讲，当一棵二叉树的每一个结点的两棵子树都具有相似的大小时，我们说树是平衡的 (balanced)。这个概念可以通过几种方式精确化。到达树的一个结点的开销和树的深度成正比，而对于平衡树，则是结点个数的对数。深度为10的完全二叉树包含了1 023个结点，最多九步内都可以到达。一棵深度为20的树可以包含超过 10^6 个结点。平衡树可以对大量的数据进行高效的访问。

143

调用 `comptree(1, n)` 可以建立一个深度为 n 的完全二叉树，它的结点标签为1到 $2^n - 1$ ：

```
fun comptree (k,n) =
  if n=0 then Lf
  else Br(k, comptree(2*k, n-1),
           comptree(2*k+1, n-1));
> val comptree = fn : int * int -> int tree
comptree (1,3);
> Br (1, Br (2, Br (4, Lf, Lf),
>           Br (5, Lf, Lf))),
>
>       Br (3, Br (6, Lf, Lf),
>
>           Br (7, Lf, Lf))) : int tree
```

`reflect` 是一个作用在树上的函数，它实现了树的镜像，办法是从上到下地交换左右子树：

```
fun reflect Lf = Lf
  | reflect (Br(v,t1,t2)) = Br(v, reflect t2, reflect t1);
> val reflect = fn : 'a tree -> 'a tree
reflect tree4;
> Br (4, Br (5, Br (7, Lf, Lf),
>           Br (6, Lf, Lf))),
>
>       Br (2, Br (3, Lf, Lf),
>
>           Br (1, Lf, Lf))) : int tree
```

练习 4.13 书写一个函数 `compsame(x, n)` 来构造一棵深度为 n 的完全二叉树，并将所有结点标签为 x 。你的函数的效率怎样？

练习 4.14 当二叉树的每个结点 $Br(x, t_1, t_2)$ 都满足 $|size(t_1) - size(t_2)| \leq 1$ 时，它就是（按大小）平衡的。最简单的检测树是否平衡的递归函数就是将 `size` 应用到每一棵子树上，但这样会进行很多冗余的计算。书写一个高效的函数来检测树是否平衡。

练习 4.15 书写一个函数来判定任意的两棵树 t 和 u 是否满足 $t = reflect(u)$ 。要求这个函数不能构造新树，因此它不能调用 `reflect` 和 `Br`，不过它可以在模式中使用 `Br`。

练习 4.16 表并没有必要内置在ML语言中。给出一个和 α list 等价的数据类型声明。

144

练习 4.17 声明一个标签二叉树的数据类型 $(\alpha, \beta)ltree$ ，其中分支结点具有类型为 α 的标签，叶子则具有类型为 β 的标签。

练习 4.18 声明一个树的数据类型，其中树的每个分支结点可以有任意有限数目的分支。（提示：使用表。）

4.11 枚举树的内容

考虑这样一个问题，就是构造一个由树的标签所组成的表。标签必须按照某种顺序排列。有三种著名的顺序，前序 (preorder)、中序 (inorder) 和后序 (postorder)，可以被树的递归函数所描述。给定一个结点，每一种顺序都是将左子树的标签放在右子树的标签前；这些顺

序仅在给定结点的标签位置上有所不同。

前序列表将给定结点的标签放在前面：

```
fun preorder Lf          = []
  | preorder (Br(v,t1,t2)) = [v] @ preorder t1 @ preorder t2;
> val preorder = fn : 'a tree -> 'a list
preorder birnam;
> ["The", "wood", "of", "Birnam"] : string list
preorder tree4;
> [4, 2, 1, 3, 5, 6, 7] : int list
```

中序列表将给定结点的标签放在左右子树标签之间，这给出了严格的由左到右的遍历：

```
fun inorder Lf          = []
  | inorder (Br(v,t1,t2)) = inorder t1 @ [v] @ inorder t2;
> val inorder = fn : 'a tree -> 'a list
inorder birnam;
> ["wood", "Birnam", "of", "The"] : string list
inorder tree4;
> [1, 2, 3, 4, 6, 5, 7] : int list
```

后序列表将给定结点的标签放在最后：

```
fun postorder Lf          = []
  | postorder (Br(v,t1,t2)) = postorder t1 @ postorder t2 @ [v];
> val postorder = fn : 'a tree -> 'a list
postorder birnam;
> ["Birnam", "of", "wood", "The"] : string list
postorder tree4;
> [1, 3, 2, 6, 7, 5, 4] : int list
```

145

虽然这些函数非常清楚，但是它们却需要四次方的时间来处理非常不平衡的树。罪魁祸首是长表的追加 (@)。通过使用额外的参数 *vs* 来积累标签，可以省掉这个操作。下面的版本对每一个分支结点恰好进行一次构造 (::) 操作：

```
fun preord (Lf, vs)      = vs
  | preord (Br(v,t1,t2), vs) = v :: preord(t1, preord(t2, vs));

fun inord (Lf, vs)      = vs
  | inord (Br(v,t1,t2), vs) = inord(t1, v::inord(t2, vs));

fun postord (Lf, vs)    = vs
  | postord (Br(v,t1,t2), vs) = postord(t1, postord(t2, v::vs));
```

这些定义是值得研究的，因为很多函数都是类似这样声明的。例如，逻辑项本质上是树，某项中的所有常量的列表就可以像上面这样建立。

练习 4.19 描述 *inorder(birnam)* 和 *inord(birnam, [])* 是怎样计算的，说明各进行了多少次构造操作。

练习 4.20 完成下面的等式，并解释其正确性。

preorder(reflect(t)) = ?

inorder(reflect(t)) = ?

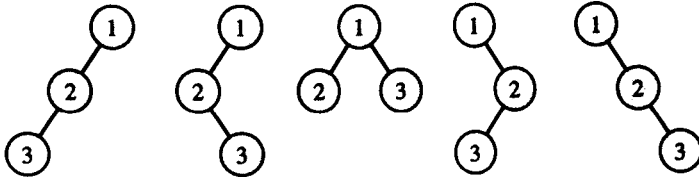
postorder(reflect(t)) = ?

4.12 由表建树

现在来考虑将放在一个表里的标签转换成一棵树。前序、中序和后序的概念也适用于这个逆操作。就算是按照同一个顺序，一个表也可以转换成很多不同的树。下面的方程

$$preorder(t) = [1, 2, 3]$$

对于 t 有5个解:



这些树里面只有一棵是平衡的。要想构造平衡树，就要将标签的列表大致分为两半。这样，两棵子树的大小（结点的个数）最多可能只差1个结点。

146

要用前序标签表来创建平衡树，就要将第一个标签放在树的根结点上:

```
fun balpre [] = Lf
  | balpre (x::xs) =
    let val k = length xs div 2
    in Br(x, balpre (List.take(xs,k)), balpre (List.drop(xs,k)))
    end;
> val balpre = fn : 'a list -> 'a tree
```

这个函数是`preorder`的逆函数。

```
balpre(explode "Macbeth");
> Br (#"M", Br (#"a", Br (#"c", Lf, Lf),
>                      Br (#"b", Lf, Lf)),
>      Br (#"e", Br (#"t", Lf, Lf),
>      Br (#"h", Lf, Lf))) : char tree
implode(preorder it);
> "Macbeth" : string
```

要用中序表来创建平衡树，就必须从中间取根结点标签。这类似于3.21节的自顶向下合并排序:

```
fun balin [] = Lf
  | balin xs =
    let val k = length xs div 2
    val y::ys = List.drop(xs,k)
    in Br(y, balin (List.take(xs,k)), balin ys)
    end;
> val balin = fn : 'a list -> 'a tree
```

这个函数是`inorder`的逆函数。

```
balin(explode "Macbeth");
> Br (#"b", Br (#"a", Br (#"M", Lf, Lf),
>                      Br (#"c", Lf, Lf)),
>      Br (#"t", Br (#"e", Lf, Lf),
>      Br (#"h", Lf, Lf))) : char tree
implode(inorder it);
> "Macbeth" : string
```

练习 4.21 书写一个函数将后序标签表转换成平衡树。

练习 4.22 函数 *balpre* 利用前序表构造了一棵树。书写一个函数，对于给定的一个标签表，构造出所有前序列表等于这个标签表的树。

147

4.13 为二叉树设计的结构

像以往一样，我们伴随着一个假想的ML会话，逐个地敲出了树的函数。现在我们应该将最重要的一些函数收集起来放进一个结构，称这个结构为 *Tree*。我们确实必须这么做，因为我们的一个函数 (*size*) 和内置的函数冲突了。使用结构的原因之一是避免这种名字冲突。

但是，我们应该把树的数据类型声明留在结构之外。如果不这样做，就要被迫使用 *Tree.Lf* 和 *Tree.Br* 来引用构造子，这会使得模式很难阅读。^① 因此，接下来我们假设已经做了下列的声明：

```
datatype 'a tree = Lf
                  | Br of 'a * 'a tree * 'a tree;

structure Tree =
struct
  fun size Lf      = 0
    | size (Br(v,t1,t2)) = 1 + size t1 + size t2;

  fun depth ...
  fun reflect ...
  fun preord ...
  fun inord ...
  fun postord ...
  fun balpre ...
  fun balin ...
  fun balpost ...
end;
```

练习 4.23 让我们把数据类型声明放进结构内，然后用下面的声明使得构造子直接对外可见：

```
val Lf = Tree.Lf;
val Br = Tree.Br;
```

这个想法错在哪里？

基于树的数据结构

148

计算机程序设计就是根据一套给定的基本操作来按需要实现一套高级操作。那些高级操作又可以成为下一个层次的程序编写的基本操作。分层的网络协议就是这个原则的一个突出的例子，在任何模块化系统设计中都可以见到这个原则。

考虑简单一点的数据结构设计。我们的任务是要基于程序设计语言提供的基本数据结构来实现所需的数据结构。这里，数据结构不仅仅是描述为它的内部实现，而且也包括它所支持的操作。为了实现新的数据结构，我们必须清楚地知道所希望的那套操作是什么。

① 有一种办法，*open* 声明，可以让一个结构的组件名字直接被看到，像 *Lf* 和 *Br* 那样。然而，打开整个 *Tree* 结构首先就破坏了声明这个结构的目的。7.14 节会讨论多种处理复合名字的办法。

ML有两个很大的优点。它的基本特性可以很简单地描述树，我们不需要担心引用和分配空间。另外，可以通过签名来描述所希望的整套操作。

关于数据集的典型操作包括插入数据、查找数据、删除数据以及合并两个数据集。我们来考虑三种可以用树来表示的数据结构：

- 字典，其中数据项是用名字来标识的。
- 数组，其中数据项是用整数来标识的。
- 优先队列，其中数据项是用优先级来标识的：只有优先级最大的项可以被删除。

和其他课本中所描述的数据结构不同，这里的数据结构是纯函数式的。插入和删除数据项不会改变数据集，而只是建立一个新的数据集。当听说这样做效率也很高的时候，你可能会觉得惊讶。

4.14 字典

字典 (dictionary) 是一些项的集合，每个数据项由唯一的键值 (通常是字符串) 来标识。它支持下面的操作：

- 查找 (lookup) 一个键值并返回相关联的数据项。
- 插入 (insert) 新的键值 (原来没有的) 以及相关的数据项。
- 更新 (update) 现有键值相关联的数据项 (如果键值不存在则进行插入)。

我们可以通过ML签名来使这个描述变得更精确：

```
signature DICTIONARY =
  sig
    type key
    type 'a t
    exception E of key
    val empty : 'a t
    val lookup : 'a t * key -> 'a
    val insert : 'a t * key * 'a -> 'a t
    val update : 'a t * key * 'a -> 'a t
  end;
```

这个签名所描述的操作要比上面描述的三个操作多。多出来的是干什么的呢？

- *key* 是搜索键值的类型。[⊖]
- *α t* 是字典的类型，里面存储的数据项具有类型 *α*。
- *E* 是当错误发生时抛出的一个异常。在找不到键值的时候，会导致查找失败，而在键值已经存在的时候，会导致插入失败。异常带有被拒绝的键值。
- *empty* 是空字典。

和这个签名匹配的结构必须声明适当类型的字典操作。例如，函数 *lookup* 需要一个字典和一个键值作为参数，并返回一个数据项。在签名 *DICTIONARY* 里并没有提到树，因此可以采用任何一种表示方法。

二叉搜索树可以实现字典。一棵合理的平衡树 (图4-1) 要比一个 (*key*, *item*) 序偶的关联表高效得多。对于表，在 *n* 个项中搜索一个键值需要 $O(n)$ 时间，而对于二叉搜索树来说则仅需要 $O(\log n)$ 时间。更新树的时间也是 $O(\log n)$ 。关联表可以在常数时间内更新，但是这并不能补偿其较长的搜索时间。

[⊖] 在这里是字符串；7.10节将会推广二叉搜索树，将搜索键值的类型作为一个参数。

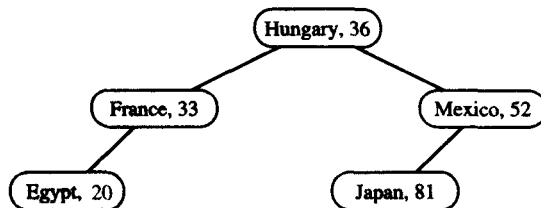


图4-1 一棵平衡的二叉搜索树

在最坏的情况下，二叉搜索树实际上要比关联表更慢。一系列的更新可能会构造出一棵很不平衡的树。搜索和更新 n 个数据项的树可能会需要 n 步。

关联表中的键值类型只要满足相等测试就可以了，但是二叉搜索树中的键值必须是线性有序的。使用按照字母顺序排序的字符串是个不错的选择。树的每一个分支结点都存放着一个 $(string, item)$ 序偶；它的左子树只存放较小的字符串；右子树则只存放较大的字符串。中序标签表将字符串按字母顺序进行排列。

与Pascal中编码的树操作不同，更新和插入操作不会改变现有的树，而是建立一棵新树。这并不像听起来那么浪费：新树共享了大部分旧树的空间。^①

图4-2展示了结构`Dict`，它是签名`DICTIONARY`的一个实例。它从声明类型`key`和`tree`以及异常`E`开始。新声明的类型仅仅是缩写，但却是必需的，主要是为了满足签名的需要。

```

structure Dict : DICTIONARY =
  struct

    type key = string;
    type 'a t = (key * 'a) tree;

    exception E of key;

    val empty = Lf;

    fun lookup (Lf, b) = raise E b
      | lookup (Br ((a,x), t1, t2), b) =
        (case String.compare(a,b) of
          GREATER => lookup(t1, b)
        | EQUAL   => x
        | LESS    => lookup(t2, b));

    fun insert (Lf, b, y) = Br((b,y), Lf, Lf)
      | insert (Br((a,x), t1, t2), b, y) =
        (case String.compare(a,b) of
          GREATER => Br((a,x), insert(t1,b,y), t2)
        | EQUAL   => raise E b
        | LESS    => Br((a,x), t1, insert(t2,b,y)));

    fun update (Lf, b, y) = Br((b,y), Lf, Lf)
      | update (Br((a,x), t1, t2), b, y) =
        (case String.compare(a,b) of
          GREATER => Br((a,x), update(t1,b,y), t2)
        | EQUAL   => Br((a,y), t1, t2)
        | LESS    => Br((a,x), t1, update(t2,b,y)));

  end;
  
```

图4-2 一个用二叉搜索树实现的字典结构

① 读者可以回忆一下3.5节的注解。——译者注

二叉搜索树中的查找是很简单的。在分支结点处，如果要查找的数据项比当前标签小则看左子树，大则看右子树。如果数据项没找到则抛出异常 E 。注意观察对于数据类型 $order$ 的使用。

插入序偶($string, item$)涉及到确定 $string$ 的正确位置，然后插入 $item$ 。就像 $lookup$ 一样，通过和当前结点的标签进行比较来确定是向左找还是向右找。这样得到的结果是一个新的分支结点；其中一棵子树被更新了，而另一个则是借用原来的。如果在树中找到了该字符串，则抛出异常。

实质上， $insert$ 是复制了从根结点到新结点的路径。函数 $update$ 除了在发现字符串的情况下结果不同，其他和 $insert$ 完全一样。

$lookup$ 中的异常是很容易去掉的，因为这个函数是迭代的。它可以返回一个类型为 $\alpha\ option$ 的结果，如果键值被找到则结果为 $SOME\ x$ ，否则结果为 $NONE$ 。而 $insert$ 里面的异常就是另外一回事了：由于那些递归调用构造了一棵新树，返回 $SOME\ t$ 或 $NONE$ 是很麻烦的。函数 $insert$ 可以先调用 $lookup$ 再调用 $update$ ，这样虽然去掉了异常，但比较次数却又加倍了。

二叉搜索树是从空树(Lf)开始，通过不断地更新和插入建立起来的。我们建立一棵树 $ctree1$ ，包括France和Egypt:

```
Dict.insert(Lf, "France", 33);
> Br (("France", 33), Lf, Lf) : int Dict.t
val ctree1 = Dict.insert(it, "Egypt", 20);
> val ctree1 = Br (("France", 33),
>                  Br (("Egypt", 20), Lf, Lf),
>                  Lf) : int Dict.t
```

再插入Hungary和Mexico:

```
Dict.insert(ctree1, "Hungary", 36);
> Br (("France", 33), Br (("Egypt", 20), Lf, Lf),
>      Br (("Hungary", 36), Lf, Lf)) : int Dict.t
Dict.insert(it, "Mexico", 52);
> Br (("France", 33), Br (("Egypt", 20), Lf, Lf),
>      Br (("Hungary", 36), Lf,
>          Br (("Mexico", 52), Lf, Lf))) : int Dict.t
```

通过插入Japan，我们建立含有5个数据项的树 $ctree2$ 。

```
val ctree2 = Dict.update(it, "Japan", 81);
> val ctree2 =
> Br (("France", 33), Br (("Egypt", 20), Lf, Lf),
>      Br (("Hungary", 36), Lf,
>          Br (("Mexico", 52),
>              Br (("Japan", 81), Lf, Lf),
>              Lf))) : int Dict.t
```

注意， $ctree1$ 仍旧存在，虽然 $ctree2$ 是由它构造的。

```
Dict.lookup(ctree1, "France");
> 33 : int
Dict.lookup(ctree2, "Mexico");
> 52 : int
Dict.lookup(ctree1, "Mexico");
> Exception: E
```

随机地插入数据项会构造出不平衡的树。如果在进行了大多数的插入操作之后，紧跟着有许多查找的话，那么在查找之前有必要将树做一下平衡。由于二叉搜索树与有序的中序列表是对应的，所以可以将它先转换成中序表，然后再由中序表来构造新树，最终实现平衡：

```
Tree.inord (ctree2, []);
> [("Egypt", 20), ("France", 33), ("Hungary", 36),
> ("Japan", 81), ("Mexico", 52)] : (Dict.key * int) list
val baltree = Tree.balin it;
> val baltree =
> Br (("Hungary", 36),
>   Br (("France", 33), Br (("Egypt", 20), Lf, Lf), Lf),
>   Br (("Mexico", 52), Br (("Japan", 81), Lf, Lf), Lf))
> : (Dict.key * int) tree
```

这就是图4-1所描绘的树。

❶ 平衡树算法。上面提到的平衡方法是有局限性的。使用`inord`和`balin`依赖于字典的内部表示是树；结果的类型现在是`tree`而不是`Dict.t`。更糟的是，使用者必须决定什么时候进行平衡。

有几种搜索树是可以自动保持平衡的，通常是在更新或查找的同时重新排列结点。Adams (1993) 展示了自平衡二叉搜索树的ML代码。Reade (1992) 展示了用函数式实现的2-3树，这种树的每个结点可以有两个或三个子结点。

练习 4.24 给出四个二叉搜索树的例子，这些树的深度为5，并只含有`ctree2`的5个标签。对于每一个例子，给出一个可以建立这棵树的插入序列。

练习 4.25 书写一个新的结构`Dict`，其中字典用 $(key, item)$ 的序偶表来表示，表元素按键值排序。

4.15 函数式数组和弹性数组

数组是什么？对于大多数程序员来说，数组是一组可更新的存储单元，这组单元以整数作为索引。传统的程序设计技巧主要是关心怎样有效地使用数组。由于多数的数组是按顺序扫描的，因此函数式程序设计员可以用表来代替数组。然而许多应用，如最简单的散列表和柱状图，是需要随机访问的。

本质上，数组是定义在有限范围整数上的一个映射。将和整数 k 关联的元素写成 $A[k]$ 。传统上，数组是通过赋值命令

$$A[k] := x$$

来更新的，这改变了机器状态使得 $A[k] = x$ 。之前 $A[k]$ 中存储的内容就不存在了。原地更新在时间和空间上都是非常高效的，但是很难与函数式程序设计相互和谐。

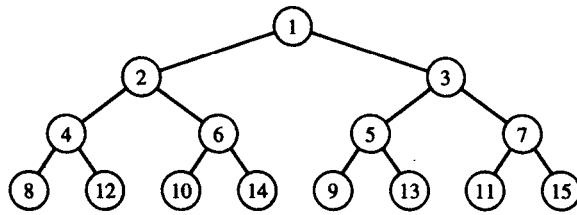
函数式数组提供了由整数到数组元素的映射，通过更新操作建立一个新的数组

$$B = \text{update } (A, k, x)$$

使得对于所有 $i \neq k$ 都有 $B[i] = A[i]$ 和 $B[k] = x$ 。数组 A 仍然存在，还可以用它创建新的数组。函数式数组可以用二叉树来实现。下标 k 在树中的位置可以这样计算：从树的根开始不断地将 k 除以2，直至简化到1为止。每一步所得的余数，如果为0则转到左子树，为1则转到右子树。

例如下标12可以通过左、左、右到达:

154



弹性 (flexible) 数组增加了在数组两端插入和删除的操作, 对通常的查询和更新操作做了扩充。程序由一个空数组开始根据需要插入数组元素。数组中不允许有空隙: 元素 $n+1$ 必须在元素 n 之后定义, 这里 $n > 0$ 。让我们考察一下底层的树操作, 它们要归功于W. Braun。

(依下标) 查询函数, *sub*, 不断地将下标除以2直到1为止。如果余数为0, 函数转向左子树, 否则转向右子树。如果到达一个叶子, 那么函数通过标准的异常来发出错误信号。

```

fun sub (Lf, _) = raise Subscript
| sub (Br(v,t1,t2), k) =
  if k = 1 then v
  else if k mod 2 = 0
    then sub (t1, k div 2)
    else sub (t2, k div 2);
> val sub = fn : 'a tree * int -> 'a
  
```

更新函数, *update*, 也是不断地将下标除以2。当到达1时, 它将分支结点替换成另一个具有新标签的分支。一个叶子可以被替换成分支结点, 这样就扩展了数组, 但必须保证不会加入吊在中间的结点, 这就足使数组没有空隙了。

```

fun update (Lf, k, w) =
  if k = 1 then Br (w, Lf, Lf)
  else raise Subscript
| update (Br(v,t1,t2), k, w) =
  if k = 1 then Br (w, t1, t2)
  else if k mod 2 = 0
    then Br (v, update(t1, k div 2, w), t2)
    else Br (v, t1, update(t2, k div 2, w));
> val update = fn : 'a tree * int * 'a -> 'a tree
  
```

调用*delete*(*ta*, *n*)将以位置*n*为根的子树 (如果存在的话) 替换成叶子。这类似*sub*, 区别在于它建立了一棵新树。

```

fun delete (Lf, n) = raise Subscript
| delete (Br(v,t1,t2), n) =
  if n = 1 then Lf
  else if n mod 2 = 0
    then Br (v, delete(t1, n div 2), t2)
    else Br (v, t1, delete(t2, n div 2));
> val delete = fn : 'a tree * int -> 'a tree
  
```

通过弹性数组的上端 (右端) 来扩展和缩小数组是很容易的。只要将上界和二叉树存储在一起并使用*update*和*delete*就行了。但是我们怎样才能从下端 (左端) 对数组进行扩展和缩小呢? 由于数组的下界是固定的, 这好像隐含了所有数组元素的移位。

考虑从下端 (根) 向一棵树增加元素 w 。结果就是 w 占据位置1, 取代了原来的根元素 v 。

新树的右子树（位置3, 5, ...）完全就是旧树的左子树（位置2, 4, ...）。通过一个递归调用，新树的左子树由 v 占据位置2，并将旧树的右子树（位置3, 5, ...）剩下的部分作为新树的左子树。

```
fun loext (Lf, w)           = Br(w, Lf, Lf)
  | loext (Br(v,t1,t2), w) = Br(w, loext(t2,v), t1);
> val loext = fn : 'a tree * 'a -> 'a tree
```

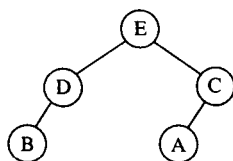
所以我们可以对数时间内从下端扩展弹性数组。要缩短数组，只要将步骤反过来就行了。试图从空数组里删除元素会抛出标准异常`Size`。形如`Br(_, Lf, Br _)`的树是不需要考虑的：在任什么时候我们都有 $L-1 < R < L$ ，其中 L 是左子树的大小， R 是右子树的大小。

```
fun lorem Lf                = raise Size
  | lorem (Br(_,Lf,Lf))      = Lf
  | lorem (Br(_, t1 as Br(v,_,_), t2)) = Br(v, t2, lorem t1);
> val lorem = fn : 'a tree -> 'a tree
```

155 该是演示的时候了。通过从一个叶子开始不断地应用`loext`，我们创建了一个逆序的从字母A到
156 E的数组：

```
loext(Lf, "A");
> Br ("A", Lf, Lf) : string tree
loext(it, "B");
> Br ("B", Br ("A", Lf, Lf), Lf) : string tree
loext(it, "C");
> Br ("C", Br ("B", Lf, Lf), Br ("A", Lf, Lf))
> : string tree
loext(it, "D");
> Br ("D", Br ("C", Br ("A", Lf, Lf), Lf),
>         Br ("B", Lf, Lf)) : string tree
val tlet = loext(it, "E");
> val tlet = Br ("E", Br ("D", Br ("B", Lf, Lf), Lf),
>                     Br ("C", Br ("A", Lf, Lf), Lf))
> : string tree
```

树`tlet`看上去是这样的：



更新`tlet`中的元素不会影响这个数组，不过建立了新的数组：

```
val tdag = update(update (tlet, 5, "Amen"),
                  2, "dagger");
> val tdag =
> Br ("E", Br ("dagger", Br ("B", Lf, Lf), Lf),
>         Br ("C", Br ("Amen", Lf, Lf), Lf))
> : string tree
sub (tdag, 5);
> "Amen" : string
sub (tlet, 5);
> "A" : string
```

这个二叉树在每个操作之后都保持平衡。对于下标为 k 的元素，其查询和更新的时间和 $\log k$ 成正比，这是任何没有大小限制的数据结构所能达到的最好时间复杂度。访问一百万个元素的数组所需的时间是访问一千个元素的数组的两倍。

标准库的结构Array提供了命令式的数组。它们将在第8章用来实现函数式（但不是弹性）数组。如果用命令式的风格使用数组，那么该实现可以提供常数时间的快速访问。命令式数组是非常流行的，以至于函数式的应用需要一些想像力。

下面是弹性数组的签名。它是基于结构Array的，但是包括了扩展和删除，既有从下端的（*loext*和*lorem*），也有从上端的（*hiext*和*hirem*）。

157

```
signature FLEXARRAY =
  sig
    type 'a array
    val empty : 'a array
    val length : 'a array -> int
    val sub : 'a array * int -> 'a
    val update : 'a array * int * 'a -> 'a array
    val loext : 'a array * 'a -> 'a array
    val lorem : 'a array -> 'a array
    val hiext : 'a array * 'a -> 'a array
    val hirem : 'a array -> 'a array
  end;
```

图4-3展示了实现。基本的树操作函数被包装在结构Braun中，以避免与结构Flex中的类似函数发生名字冲突。不巧的是，Braun里面的下标范围是从1到 n ，而Flex的下标则是从0到 $n-1$ 。前者是来自数据表示方法，后者则是ML的约定。

结构Flex将弹性数组表达为二叉树和整数的序偶，其中，整数表示了数组的大小。也许可以将类型array声明为类型缩写：

```
type 'a array = 'a tree * int;
```

不过，实际上却将array声明成了具有一个构造子的数据类型。这种数据类型并不会增加运行时的开销：那个唯一的构造子是不占空间的。这个新的类型将弹性数组同碰巧的树和整数的序偶区分开来，在调用Braun.sub时出现的那个序偶就是这样。这个构造子在结构之外是不可见的，防止了使用者将一个弹性数组分解开来。

① 进一步的阅读。Dijkstra (1976)，是一本经典的关于命令式程序设计的著作，介绍了弹性数组和很多其他概念。Hoogerwoord (1992) 详细叙述了弹性数组，包括从下端扩展数组的操作。Okasaki (1995) 介绍了随机访问表 (random access list)，它既有对数时间的数组式的访问，也有常数时间的表操作（构造、表头、表尾）。随机访问表是由一组完全二叉树的列表来表示的。它的代码是用ML写的，并且非常易懂。

练习 4.26 书写一个函数，创建一个其下标位置从1到 n 都是含有元素 x 的数组。不要使用Braun.update，直接创建这棵树。

练习 4.27 书写一个函数将由元素 x_1, x_2, \dots, x_n （分别在下标位置1到 n ）组成的数组转换成一个表。直接对树进行操作，而不是反复地使用下标进行操作。

练习 4.28 通过允许空标签出现在树中来实现稀疏数组，这种数组中间可以有很大的空隙。

```

structure Braun =
  struct
    fun sub    ...
    fun update ...
    fun delete ...
    fun loext  ...
    fun lorem  ...
  end;

structure Flex : FLEXARRAY =
  struct
    datatype 'a array = Array of 'a tree * int;

    val empty = Array(Lf, 0);

    fun length (Array(_, n)) = n;

    fun sub (Array(t, n), k) =
      if 0 <= k andalso k < n then Braun.sub(t, k+1)
      else raise Subscript;

    fun update (Array(t, n), k, w) =
      if 0 <= k andalso k < n then Array(Braun.update(t, k+1, w), n)
      else raise Subscript;

    fun loext (Array(t, n), w) = Array(Braun.loext(t, w), n+1);

    fun lorem (Array(t, n)) =
      if n > 0 then Array(Braun.lorem t, n-1)
      else raise Size;

    fun hiext (Array(t, n), w) = Array(Braun.update(t, n+1, w), n+1);

    fun hirem (Array(t, n)) =
      if n > 0 then Array(Braun.delete(t, n), n-1)
      else raise Size;

  end;

```

图4-3 Braun树和弹性数组的结构

4.16 优先队列

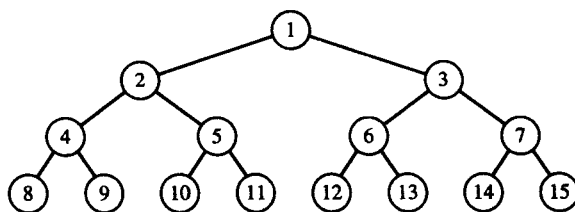
优先队列 (priority queue) 是有序的数据项集。数据项可以以任意顺序插入，但是只有最高优先级的项可以被访问和删除。传统上较小的数值代表了较高的优先级，所以有关的基本操作叫做 *insert* (插入)、*min* (访问最小项) 和 *delmin* (删除最小项)。

在仿真中，优先队列可以根据调度时间来选出下一个事件。在人工智能中，优先队列实现了最佳优先搜索 (best-first search)：对一个问题的候选解法按优先级别来存储 (由一个评

估函数给每个解法赋予优先级别), 最好的候选解法被选出来进入下一步的搜索。

如果优先队列是存放在排好序的表中, 那么对于有 n 个元素的队列, 插入操作最坏需要 n 步。这会慢得不能接受。通过二叉树, 插入一项和删除最小项只需要 $O(\log n)$ 步。这样的树叫做堆 (heap), 是著名的堆排序 (heap sort) 算法的基础。树中标签的安排要满足每一个标签都不比它上面的标签小。这个堆条件 (heap condition) 并不能使得标签严格按顺序排列, 但是一定会将会最小的标签放在根结点上。

传统上, 这棵树被嵌在一个数组里, 标签是如下图那样对应于数组下标的:



n 项的堆由结点1到 n 组成。这种索引模式总是建立高度尽可能小的树。不过我们的函数式优先队列是基于实现弹性数组的那种索引模式; 它似乎更适合函数式程序设计, 并且也保证了最小高度。最后得到的程序是新旧思想的混合。

如果堆包含 $n-1$ 项, 那么一个插入项将填充位置 n 。但是, 新的项可能太小了, 不能在保持堆条件的前提下放在那个位置上。它可能最后占据树中高一些的位置, 迫使比它大的项向下移动。函数`insert`和`loexr`的原理是一样的, 只不过在插入时要保持堆条件。当新项 w 不超过当前标签 v 时, w 会取代 v 成为当前标签, 而把 v 往下插入到子树中去; 因为 v 不会比子树中的项大, 所以 w 也不会。

```

fun insert(w: real, Lf)      = Br(w, Lf, Lf)
  | insert(w, Br(v, t1, t2)) =
    if w <= v then Br(w, insert(v, t2), t1)
    else Br(v, insert(w, t2), t1);
> val insert = fn : real * real tree -> real tree
  
```

看上去没有什么简单的办法将`insert`操作倒过来。删除必须去掉在根结点的项, 因为它是最小的。但是从 n 项的堆中删除一项以后必须空出位置 n 。项 n 显然可能太大, 而不能在保持堆条件的前提下放进根结点。因此, 我们需要两个函数: 一个是要删除项 n , 另一个是将它的标签重新插入到合适的位置。

函数`leftrem`和`lorem`原理一样, 但是并不会将标签在堆中上下移动。它总是删除树最左边的一项, 并将子树交换以保持堆的正确形状。它返回被删除的项和新的堆所组成的序偶:

```

fun leftrem (Br(v, Lf, Lf)) = (v, Lf)
  | leftrem (Br(v, t1, t2)) =
    let val (w, t) = leftrem t1
    in (w, Br(v, t2, t)) end;
> val leftrem = fn : 'a tree -> 'a * 'a tree
  
```

函数`siftdown`用被转移的项 (`leftrem`删掉的项) 和旧堆的两个子树构造一个新堆。这个项在树中向下移动。在每一个分支处, 它转向有着较小标签的那棵子树。如果没有比它的标签还小的子树, 就停止移动了。我们要感谢这个索引模式, 下面代码中没考虑的情形是不可能发生

的。如果左子树是空的，那么右子树也一定是空的；如果右子树是空的，那么左子树只能有一个结点。

```
fun siftdown (w:real, Lf, Lf)          = Br(w,Lf,Lf)
  | siftdown (w, t as Br(v,Lf,Lf), Lf) =
    if w <= v then Br(w, t, Lf)
    else Br(v, Br(w,Lf,Lf), Lf)
  | siftdown (w, t1 as Br(v1,p1,q1), t2 as Br(v2,p2,q2)) =
    if w <= v1 andalso w <= v2 then Br(w,t1,t2)
    else if v1 <= v2 then Br(v1, siftdown(w,p1,q1), t2)
    (*v2 < v1*) else Br(v2, t1, siftdown(w,p2,q2));
> val siftdown = fn
> : real * real tree * real tree -> real tree
```

现在可以完成删除了。函数`delmin`调用`leftrem`来删除和返回一项，然后`siftdown`把这项放回一个合适的位置。从空的堆里面删除是错误的，并且要单独处理仅有一项的堆。

```
fun delmin Lf          = raise Size
  | delmin (Br(v,Lf,_)) = Lf
  | delmin (Br(v,t1,t2)) =
    let val (w,t) = leftrem t1
    in siftdown (w,t2,t) end;
> val delmin = fn : real tree -> real tree
```

161 为优先队列而定的签名描述了上面讨论的基本操作。它也描述了将堆和表相互转换的操作，以及相应的排序函数。签名将队列里面的项的类型描述为`item`；目前这个类型是`real`，不过一个函子可以将任何有序的类型作为参数（见7.10节）。

```
signature PRIORITY_QUEUE =
sig
  type item
  type t
  val empty   : t
  val null    : t -> bool
  val insert  : item * t -> t
  val min     : t -> item
  val delmin  : t -> t
  val fromList : item list -> t
  val toList  : t -> item list
  val sort    : item list -> item list
end;
```

图4-4显示了关于堆的结构。它给空堆`empty`和谓词`null`使用了显然的定义。函数`min`只是简单地返回根结点的标签。

优先队列很容易就实现了堆排序。将表排序只是将它转换成堆然后再转换回来。函数`heapify`将表转换成堆的方法使我们想起了自顶向下的合并排序（3.21节）。这个方法，通过`siftdown`，在线性时间内将堆构造出来；反复地插入则需要 $O(n \log n)$ 时间。堆排序的时间复杂度是最理想的：它在最坏的情况下也只需要 $O(n \log n)$ 时间来排序 n 个项目。实践中，堆排序往往比其他的 $O(n \log n)$ 算法要慢。记得我们在第3章的时间实验，快速排序和合并排序可以在200毫秒以内排序10 000个随机数，但是`Heap.sort`则要500毫秒。

```

structure Heap : PRIORITY_QUEUE =
  struct
    type item = real;
    type t    = item tree;

    val empty = Lf;

    fun null Lf      = true
      | null (Br _) = false;

    fun min (Br(v,_,_)) = v;

    fun insert ...

    fun leftrem ...

    fun siftdown ...

    fun delmin ...

    fun heapify (0, vs)      = (Lf, vs)
      | heapify (n, v::vs) =
        let val (t1, vs1) = heapify (n div 2, vs)
            val (t2, vs2) = heapify ((n-1) div 2, vs1)
        in (siftdown (v,t1,t2), vs2) end;

    fun fromList vs = #1 (heapify (length vs, vs));

    fun toList (t as Br(v,_,_)) = v :: toList(delmin t)
      | toList Lf                = [];

    fun sort vs = toList (fromList vs);

  end;

```

图4-4 利用堆实现的优先队列结构

虽然有其他更好的排序方法，但是堆却实现了非常理想的优先队列。为了看看堆是怎样工作的，让我们建立一个堆，并从中删除一些项。

```

Heap.fromList [4.0, 2.0, 6.0, 1.0, 5.0, 8.0, 5.0];
> Br (1.0, Br (2.0, Br (6.0, Lf, Lf),
>               Br (4.0, Lf, Lf)),
>      Br (5.0, Br (8.0, Lf, Lf),
>      Br (5.0, Lf, Lf))) : Heap.t
Heap.delmin it;
> Br (2.0, Br (5.0, Br (8.0, Lf, Lf),
>               Br (5.0, Lf, Lf)),
>      Br (4.0, Br (6.0, Lf, Lf), Lf)) : Heap.t
Heap.delmin it;
> Br (4.0, Br (6.0, Br (8.0, Lf, Lf), Lf),
>      Br (5.0, Br (5.0, Lf, Lf), Lf)) : Heap.t

```

我们看到最小的项被首先删除。我们再应用两次`delmin`:

```

Heap.delmin it;
> Br (5.0, Br (5.0, Br (8.0, Lf, Lf), Lf),
>      Br (6.0, Lf, Lf)) : Heap.t
Heap.delmin it;
> Br (5.0, Br (6.0, Lf, Lf), Br (8.0, Lf, Lf)) : Heap.t

```

ML的输出被适当地缩进以突出二叉树的结构。

❶ 其他形式的优先队列。这里展示的堆有时又叫做二叉 (binary) 堆或隐式 (implicit) 堆。像Sedgewick (1988) 这样的算法课本里面有详细的叙述。其他传统的优先队列的表示方法也可以进行函数式的编码。左倾 (leftist) 堆 (Knuth, 1973, 151页) 和二项式堆 (Cormen等, 1990, 400页) 要比二叉堆复杂得多。但是它们允许两个堆在对数时间内合并。二项式堆的合并操作是通过一种二进制加法来进行的。Chris Okasaki提供了本节的大部分代码, 他也实现了很多其他形式的优先队列。只要不要求合并操作, 二叉堆就是最简单的、最快的。

练习 4.29 画图表示堆的创建, 从空堆开始, 然后插入4、2、6、1、5、8和5 (就像上面调用 *Heap.fromList* 用的那个表)。

练习 4.30 通过用二进制表示下标来叙述函数式数组的索引模式。同样叙述传统的堆排序的索引模式。

练习 4.31 书写函数式数组的查询和更新操作, 这个函数式数组要基于传统的堆排序的索引模式。它们和 *Braun.sub*、*Braun.update* 比较怎样?

重言式检测器

这一节介绍基本的定理证明。我们定义命题和将其转换成各种范式的函数, 由此得到一个命题逻辑的重言式检测器。我们不再使用二叉树, 而是声明一个关于命题的数据类型。

4.17 命题逻辑

命题逻辑是处理命题 (proposition) 的, 命题是由原子 a, b, c, \dots 通过连接词 \wedge, \vee, \neg 构成。命题可以是

$\neg p$ 否定, “非 p ”
 $p \wedge q$ 合取, “ p 且 q ”
 $p \vee q$ 析取, “ p 或 q ”

命题类似于布尔表达式, 我们把它表示为数据类型 *prop*:

```

datatype prop = Atom of string
              | Neg of prop
              | Conj of prop * prop
              | Disj of prop * prop;

```

蕴涵 $p \rightarrow q$ 等价于 $(\neg p) \vee q$ 。下面是构造蕴涵的函数:

```

fun implies(p, q) = Disj(Neg p, q);
> val implies = fn : prop * prop -> prop

```

我们的例子基于一些重要的属性，它们是富有的 (rich)、有土地的 (landed) 和圣洁的 (saintly):

```
val rich    = Atom "rich"
and landed = Atom "landed"
and saintly = Atom "saintly";
```

下面是对于富有的、有土地的和圣洁的两个假设。

- 假设1是 $landed \rightarrow rich$: 有土地就会富有。
- 假设2是 $\neg(saintly \wedge rich)$: 一个人不可能既富有又圣洁。

一个可能的结论是 $landed \rightarrow \neg saintly$: 有土地的人不圣洁。

让我们把假设和希望的结论输入给ML:

```
val assumption1 = implies(landed, rich)
and assumption2 = Neg (Conj (saintly, rich));
> val assumption1 = Disj (Neg (Atom "landed"),
>                         Atom "rich") : prop
> val assumption2 = Neg (Conj (Atom "saintly",
>                             Atom "rich")) : prop
val concl = implies(landed, Neg saintly);
> val concl = Disj (Neg (Atom "landed"),
>                  Neg (Atom "saintly")) : prop
```

如果可以从假设导出结论，那么下面的命题就应该是一个命题的定理——重言式 (tautology)。

让我们来把它声明成要证明的目标:

```
val goal = implies (Conj (assumption1, assumption2), concl);
> val goal =
> Disj (Neg (Conj (Disj (Neg (Atom "landed"),
>                         Atom "rich"),
>                     Neg (Conj (Atom "saintly",
>                                 Atom "rich")))),
>       Disj (Neg (Atom "landed"), Neg (Atom "saintly")))
> : prop
```

用数学记法就是

$$((landed \rightarrow rich) \wedge \neg(saintly \wedge rich)) \rightarrow (landed \rightarrow \neg saintly)$$

为了显示更具可读性的结果，我们来声明一个转换命题到字符串的函数。

```
fun show (Atom a)      = a
  | show (Neg p)       = "~" ^ show p ^ " "
  | show (Conj (p,q))  = "(" ^ show p ^ " & " ^ show q ^ " "
  | show (Disj (p,q))  = "(" ^ show p ^ " | " ^ show q ^ " "
> val show = fn : prop -> string
```

下面是我们的证明目标:

```
show goal;
> "(((~(((~landed) | rich) & (~saintly & rich))))
>   | (((landed) | (~saintly))))" : string
```

如本书其他地方一样，我们插入了空格和换行以使结果更清楚。

练习 4.32 书写另一个版本的`show`，去掉不必要的括号。如果 \neg 具有最高优先级，而 \vee 具有最低的，那么 $((\neg a) \wedge b) \vee c$ 中的所有括号都是多余的。由于 \wedge 和 \vee 满足结合律，因此要去掉 $(a \wedge b) \wedge (c \wedge d)$ 中的括号。

练习 4.33 书写一个用标准的真值表来计算一个命题的函数。其中一个参数应该是取值为真的原子的列表，其他的原子则设为假。

4.18 否定范式

任何命题都可以被转换成否定范式 (negation normal form) (NNF)，其中 \neg 只应用在原子之上，这可以通过把否定推进合取式或析取式来完成。重复地进行替换

$$\begin{aligned}\neg\neg p &\text{ 替换成 } p \\ \neg(p \wedge q) &\text{ 替换成 } (\neg p) \vee (\neg q) \\ \neg(p \vee q) &\text{ 替换成 } (\neg p) \wedge (\neg q)\end{aligned}$$

这样的替换有时也称作重写规则 (rewrite rule)。首先要考虑它们对不对。规则有歧义吗？没有，因为规则的左边概括了不同的情形。替换会最终结束吗？会的，虽然会创建新的否定式，但是被否定的部分缩小了。我们怎么能知道什么时候停下来呢？在这里只要对命题扫描一遍就足够了。

函数`nnf`原封不动地应用了上述规则。当某种情形没有适当的规则时，就简单地递归调用。

```
fun nnf (Atom a)           = Atom a
  | nnf (Neg (Atom a))     = Neg (Atom a)
  | nnf (Neg (Neg p))      = nnf p
  | nnf (Neg (Conj(p,q))) = nnf (Disj(Neg p, Neg q))
  | nnf (Neg (Disj(p,q))) = nnf (Conj(Neg p, Neg q))
  | nnf (Conj(p,q))        = Conj(nnf p, nnf q)
  | nnf (Disj(p,q))        = Disj(nnf p, nnf q);
> val nnf = fn : prop -> prop
```

假设2, $\neg(\text{saintly} \wedge \text{rich})$ ，被转换成 $\neg\text{saintly} \vee \neg\text{rich}$ 。用函数`show`显示转换结果。

```
nnf assumption2;
> Disj (Neg (Atom "saintly"), Neg (Atom "rich")) : prop
show it;
> "((~saintly) | (~rich))" : string
```

函数`nnf`还可以改进。给定 $\neg(p \wedge q)$ ，函数会对

$$\text{nnf}(\text{Disj}(\text{Neg } p, \text{Neg } q))$$

求值，后面的递归调用会接着计算

$$\text{Disj}(\text{nnf}(\text{Neg } p), \text{nnf}(\text{Neg } q))$$

让函数在此情形下直接对这个表达式求值可以省去一次递归调用，同样的改进也适用于 $\neg(p \vee q)$ 。

它还可以更快。用另一个函数去计算`nnf(Neg p)`避免了不必要的否定式构造。通过相互递归，函数`nnfpos`计算`p`的范式，而函数`nnfneg`则计算`Neg p`的范式。

```

fun nnfpos (Atom a)    = Atom a
  | nnfpos (Neg p)      = nnfneg p
  | nnfpos (Conj(p,q)) = Conj(nnfpos p, nnfpos q)
  | nnfpos (Disj(p,q)) = Disj(nnfpos p, nnfpos q)
and nnfneg (Atom a)    = Neg (Atom a)
  | nnfneg (Neg p)      = nnfpos p
  | nnfneg (Conj(p,q)) = Disj(nnfneg p, nnfneg q)
  | nnfneg (Disj(p,q)) = Conj(nnfneg p, nnfneg q);

```

4.19 合取范式

合取范式是我们重言式检测器的基础，也是定理证明归结方法的基础。硬件设计师们称它为布尔表达式的极大项表示。

167

文字 (literal) 是一个原子或原子的否定式。当一个命题形如 $p_1 \wedge \dots \wedge p_m$ 时被称为合取范式 (conjunctive normal form) (CNF)，其中每个 p_i 都是文字的析取。

要检测 p 是否是重言式，把它简化成 CNF 命题。现在，如果 $p_1 \wedge \dots \wedge p_m$ 是重言式的话，那么 p_i 也是重言式，其中 $i = 1, \dots, m$ 。假设 p_i 是 $q_1 \vee \dots \vee q_n$ ，其中 q_1, \dots, q_n 都是文字。如果这些文字里面包含一个原子和它的否定式，那么 p_i 就是重言式。否则这些原子可以被赋予一套真值使得 p_i 中的每个文字都为假 (即存在成假指派)，故 p 就不是重言式。

要得到 CNF，我们先从命题的否定范式开始。利用分配律，不断将析取推进去，直到析取仅应用于文字为止。

$p \vee (q \wedge r)$ 替换成 $(p \vee q) \wedge (p \vee r)$

$(q \wedge r) \vee p$ 替换成 $(q \vee p) \wedge (r \vee p)$

这些替换并没有产生否定范式的那些来得直接。它们还是有歧义的，两条规则都适用于 $(a \wedge b) \vee (c \wedge d)$ ，所得的范式在逻辑上却是等价的。替换过程的终止是有保证的，虽然每个替换都会使命题增长，但是却把一个析取替换成几个更小的析取，这个过程不可能永远进行下去。

一个析取式可能包括内嵌的合取式；例如 $a \vee (b \vee (c \wedge d))$ 。我们的替换策略是，给定 $p \vee q$ ，首先把 p 和 q 转换成 CNF。这使得任何的合取都在最外层。然后，应用上面的替换规则，将析取分配进合取式。

调用 $distrib(p, q)$ 计算析取式 $p \vee q$ 的 CNF，前提是 p 和 q 都是 CNF。如果两者都不是合取式，那么结果就是 $p \vee q$ ，这是仅有的 $distrib$ 创建析取式的情况。其他情况下，它将析取分配进合取式。

```

fun distrib (p, Conj(q,r)) = Conj(distrib(p,q), distrib(p,r))
  | distrib (Conj(q,r), p) = Conj(distrib(q,p), distrib(r,p))
  | distrib (p, q)         = Disj(p,q) (* 没有合取式 *)
> val distrib = fn : prop * prop -> prop

```

前两个情形是有重叠的：如果 p 和 q 都是合取式，那么 $distrib(p, q)$ 匹配第一种情形，这是因为 ML 是按顺序进行模式匹配的。这是一种自然的表达函数的方法。我们可以看到， $distrib$ 从参数的各个部分中建立了所有可能的析取：

```

distrib (Conj(rich,saintly), Conj(landed, Neg rich));
> Conj (Conj (Disj (Atom "rich", Atom "landed"),
>               Disj (Atom "saintly", Atom "landed")),
>       Conj (Disj (Atom "rich", Neg (Atom "rich")),
>             Disj (Atom "saintly", Neg (Atom "rich"))))
> : prop
show it;
> "(((rich | landed) & (saintly | landed)) &
>   ((rich | (~rich)) & (saintly | (~rich))))" : string

```

$p \wedge q$ 的合取范式就是 p 和 q 分别的合取范式的合取。函数 *cnf* 是很简单的，因为 *distrib* 已经做了大部分的工作。第三个情形是为了捕捉单独的 *Atom* 和 *Neg* 的。

```

fun cnf (Conj(p,q)) = Conj (cnf p, cnf q)
  | cnf (Disj(p,q)) = distrib (cnf p, cnf q)
  | cnf p           = p      (* 这是一个文字 *);
> val cnf = fn : prop -> prop

```

最后我们利用 *cnf* 和 *nnf* 来将要证明的目标转换成 CNF:

```

val cgoal = cnf (nnf goal);
> val cgoal = Conj ( ... , ... ) : prop
show cgoal;
> "((((landed | saintly) | ((~landed) | (~saintly))) &
>   (((~rich) | saintly) | ((~landed) | (~saintly)))) &
>   (((landed | rich) | ((~landed) | (~saintly))) &
>     (((~rich) | rich) | ((~landed) | (~saintly)))))"
> : string

```

这是个不折不扣的重言式。四个析取式中的每一个都含有某个原子和它自己的否定：分别是 *landed*、*saintly*、*landed* 和 *rich*。为检测这个，函数 *positives* 返回了一个析取式中的正原子列表，而函数 *negatives* 则返回了否定原子的列表。没预测到的情形说明了命题不是一个 CNF；这时要抛出异常作为结果。

```

exception NonCNF;
fun positives (Atom a)      = [a]
  | positives (Neg (Atom _)) = []
  | positives (Disj(p,q))   = positives p @ positives q
  | positives _             = raise NonCNF;
> val positives = fn : prop -> string list
fun negatives (Atom _)      = []
  | negatives (Neg (Atom a)) = [a]
  | negatives (Disj(p,q))   = negatives p @ negatives q
  | negatives _             = raise NonCNF;
> val negatives = fn : prop -> string list

```

函数 *taut* 对任一个 CNF 命题进行重言式检测，利用了 *inter*（见 3.15 节）来形成正原子和否定原子集合的交集。最后的结论大概没那么有意思了。

```

fun taut (Conj(p,q)) = taut p andalso taut q
  | taut p           = not (null (inter (positives p, negatives p)));
> val taut = fn : prop -> bool
taut cgoal;
> true : bool

```

168
169

i 先进的重言式检测器。上面叙述的重言式检测器并不实用。有序二叉决策图 (ordered binary decision diagram, OBDD) 可以通过硬件设计来解决复杂的问题。它们利用了有向图, 每一个结点表示了一个 “if-then-else” 决策。Moore (1994) 阐述了这个思想和关键的优化办法, 其中涉及到了散列和缓冲技术。

Davis-Putnam过程使用了CNF。它可以解决很难的约束满足问题, 并解决了一些组合数学中的开放问题。Zhang和Stickel (1994) 描述了一个可以用ML编码的算法。Uribe和Stickel (1994) 也讲述了对这个过程和OBDD的实验性的比较。

练习 4.34 合取范式命题可以表示为文字表的表。外面的表是文字的合取; 每个里面的表都是文字的析取。书写一个函数将命题转换成这种表示法的CNF。

练习 4.35 改写distrib的定义, 使得各种情形没有重叠。

练习 4.36 当一个命题形如 $p_1 \vee \cdots \vee p_m$ 时被称为析取范式 (disjunctive normal form) (DNF), 其中每个 p_i 都是文字的合取。当一个命题的否定是重言式时, 这个命题就是矛盾的 (inconsistent)。叙述一种方法利用DNF来测试一个命题是否是矛盾的。用ML编写这个方法。

要点小结

- 数据类型声明是通过组合现有类型来创建新类型的。
- 模式是由构造子和变量构成的。
- 异常是响应运行时错误的一般机制。
- 递归的数据类型声明可以定义树。
- 二叉树可以表示很多数据结构, 包括字典、函数式数组和优先队列。
- 模式匹配可以表达逻辑公式的变换。

第5章 函数和无穷数据

函数式程序设计中最有力的技术就是将函数作为数据看待。大多数的函数式语言赋予函数数值完整的功能，没有强加的限制。就像其他种类的值一样，函数可以作为别的函数的参数和返回值，并且可以放在序偶、表和树里面。

过程式语言像Fortran和Pascal也接受这种思想，只要编译器的作者们觉得方便实现。函数可以是参数：比如说排序中的比较函数，或一个有待积分的数值函数。即使是这种受限制的情形也是很重要的。

如果一个函数是作用在其他函数上的话，那么它就是高阶的（higher-order）（或称算子，functional），例如算子`map`将一个函数应用到表中的每个元素上，由此建立一个新表。足够丰富的算子集合可以在不需要变量的情况下表达所有的函数。算子可以用来构造语法分析器（见第9章）以及定理证明策略（见第10章）。

无穷表，它的元素在需要的时候才会进行求值，也可以通过把函数作为数据来实现。一个惰性表的表尾是个函数，当这个函数被调用时会产生另一个惰性表。惰性表可以是无限长的，而它的任何有限数目的元素都是可以求值的。

本章提要

前半部分讲述了把函数作为数据的基本程序设计技术。后半部分则作为扩展提供了实践的例子。惰性表可以在ML（虽然有严格的求值规则）里通过函数值表示。

本章包括以下几节：

- 作为值的函数。`fn`记法可以表达一个函数而无须将其命名。任何有两个参数的函数都可以表达成只有一个参数的“柯里”函数，这个函数的结果是另一个函数。高阶函数的简单例子有多态排序函数和数值算子。
- 通用算子。高阶函数式程序设计的内容很大程度上是对某些著名算子的使用，这些算子是操作在表和其他递归数据类型之上的。
- 序列（sequence），或无穷表。本章通过一些标准的例子来展示在ML里是如何获得惰性求值机制的。更难一点的问题是将多个整数列表的列表合并成一个整数列表，当输入的整数列表是无穷表时，它们必须被合理地组合才能不丢失任何整数。
- 搜索策略和无穷表。一个搜索问题的解集合有可能是无穷的，这样的解集合可以作为无穷表来产生。解的使用者可以单独设计，使得它们独立于解的生成者，这样解的生成可以不受限制地采用任何适当的搜索策略。

171

作为值的函数

在ML里面函数是抽象的值：它们可以被创建；可以被应用到参数上；可以成为其他数据结构的一部分。除此之外都是不允许的。函数是由模式和表达式给出的，但被看作是把参数

转换成结果的“黑盒子”。

5.1 使用fn记法的匿名函数

ML的函数不一定需要名字。如果 x 是一个变量（具有类型 σ ），并且 E 是一个表达式（具有类型 τ ），那么表达式

$$\text{fn } x \Rightarrow E$$

表示了一个具有类型 $\sigma \rightarrow \tau$ 的函数。函数的参数是 x ，函数体是 E 。模式匹配也是允许的：表达式

$$\text{fn } P_1 \Rightarrow E_1 \mid \dots \mid P_n \Rightarrow E_n$$

表示了由模式 P_1, \dots, P_n 定义的函数。它和下面的let表达式具有相同的含义

$$\text{let fun } f(P_1) = E_1 \mid \dots \mid f(P_n) = E_n \text{ in } f \text{ end}$$

条件是 f 不能出现在表达式 E_1, \dots, E_n 中。fn语法不能表达递归函数。

例如， $\text{fn } n \Rightarrow n*2$ 是一个将整数翻倍的函数，它可以被应用到一个参数上；通过val声明，也可以给它命名。

```
(fn n=>n*2)(9);
> 18 : int
val double = fn n=>n*2;
> val double = fn : int -> int
```

很多ML里的构造都是通过fn记法来定义的。条件表达式

$$\text{if } E \text{ then } E_1 \text{ else } E_2$$

就是下面函数应用的缩写

$$(\text{fn } \text{true} \Rightarrow E_1 \mid \text{false} \Rightarrow E_2) (E)$$

分情表达式也可以类似地翻译。

练习 5.1 用fn记法表达下列函数。

```
fun square(x) : real = x*x;
fun cons (x,y) = x::y;
fun null [] = true
  | null (_:_) = false;
```

练习 5.2 修改下面的函数定义，用val声明代替fun声明。

```
fun area (r) = pi*r*r;
fun title(name) = "The Duke of " ^ name;
fun lengthvec (x,y) = Math.sqrt(x*x + y*y);
```

5.2 柯里函数

函数只能有一个参数。迄今，有多个参数的函数是把它们当作一个元组输入的。多个参数也可以实现为一个函数返回另一个函数作为结果。这种机制叫做柯里化（currying），是根

据逻辑学家H. B. Curry命名的。[⊖]考虑下面的函数

```
fun prefix pre =
  let fun cat post = pre ^ post
      in cat end;
> val prefix = fn : string -> (string -> string)
```

如果使用fn记法, *prefix*就是函数

```
fn pre => (fn post => pre ^ post)
```

给定一个字符串*pre*, 函数*prefix*的结果是另一个函数, 这个函数把*pre*连接在它自己的参数之前。例如, *prefix* "Sir"是函数

```
fn post => "Sir " ^ post
```

它可以被用到字符串上:

```
prefix "Sir ";
> fn : string -> string
it "James Tyrrell";
> "Sir James Tyrrell" : string
```

省略*it*, 两个函数应用可以同时做:

```
(prefix "Sir ") "James Tyrrell";
> "Sir James Tyrrell" : string
```

这其实是一个函数调用, 其中的函数是通过表达式*prefix* "Sir"计算出来的。

请注意, *prefix*用起来就像是两个参数的函数。这是个柯里函数 (curried function)。假设需要声明有两个参数的函数, 参数类型分别为 σ_1 和 σ_2 , 结果类型是 τ , 现在我们有两种途径, 一个是作用在序偶上的函数, 具有类型 $(\sigma_1 \times \sigma_2) \rightarrow \tau$; 另一个是柯里函数, 具有类型 $\sigma_1 \rightarrow (\sigma_2 \rightarrow \tau)$ 。

柯里函数允许部分应用 (partial application)。应用到它的第一个参数 (具有类型 σ_1) 时, 结果是类型为 $\sigma_2 \rightarrow \tau$ 的函数。这个函数可能是通用的: 比如说, 用于称呼所有的骑士。

```
val knightify = prefix "Sir ";
> val knightify = fn : string -> string
knightify "William Catesby";
> "Sir William Catesby" : string
knightify "Richard Ratcliff";
> "Sir Richard Ratcliff" : string
```

也可以类似地称呼其他著名的历史人物:

```
val dukify = prefix "The Duke of ";
> val dukify = fn : string -> string
dukify "Clarence";
> "The Duke of Clarence" : string
val lordify = prefix "Lord ";
> val lordify = fn : string -> string
lordify "Stanley";
> "Lord Stanley" : string
```

[⊖] 这个方法曾经归功于Schönfinkel, 但是Schönfinkeling一词从未流行过。

柯里函数的语法。上面的函数是用`val`声明的，没有用`fun`。`fun`声明必须具有显式的参数。

174 对于柯里函数来说，可以有多个参数，参数之间用空白隔开。下面是`prefix`的一个等价声明：

```
fun prefix pre post = pre ^ post;
> val prefix = fn : string -> (string -> string)
```

函数调用具有形式 $E E_1$ ，其中 E 是表示函数的表达式。由于

$E E_1 E_2 \cdots E_n$ 是 $(\cdots((E E_1) E_2) \cdots) E_n$ 的简写

因此，可以直接书写`prefix "Sir" "James Tyrrell"`，而不用括号。这个表达式是从左到右计算的。

`prefix`的类型， $string \rightarrow (string \rightarrow string)$ ，可以省略其中的括号：符号 \rightarrow 是右结合的。

递归。柯里函数也可以是递归的。调用`replist n x`构造了一个由 n 个 x 的拷贝所构成的表。

```
fun replist n x = if n=0 then [] else x :: replist (n-1) x;
> val replist = fn : int -> 'a -> 'a list
replist 3 true;
> [true, true, true] : bool list
```

即使是柯里化，递归也是按照通常的计算规则进行的。`replist 3`的结果是函数

```
fn x => if 3=0 then [] else x :: replist (3-1) x
```

将它应用到`true`上面产生了表达式

```
true :: replist 2 true
```

随着计算的继续，两个进一步的递归调用产生了

```
true :: true :: true :: replist 0 true
```

最后的调用返回`nil`，因而总的结果是`[true, true, true]`。

i 和数组的类比。在元组和柯里化之间的选择很类似于Pascal中的二维数组和嵌套数组之间的选择。

```
A: array [1..20, 1..30] of integer
B: array [1..20] of array [1..30] of integer
```

前面数组的下标形式是 $A[i, j]$ ，后者则是 $B[i][j]$ 。嵌套数组允许部分下标： $B[i]$ 是一个一维数组。[⊖]

175 练习 5.3 下面的几个柯里函数的部分应用的结果函数分别是什么？（不要在机器上试。）

```
fun plus i j : int = i+j;
fun lesser a b : real = if a<b then a else b;
fun pair x y = (x,y);
fun equals x y = (x=y);
```

练习 5.4 下面的两个函数 f 的声明有什么实际区别吗？假设已知函数 g 和柯里函数 h 。

```
fun f x y = h (g x) y;
fun f x = h (g x);
```

⊖ 有些Pascal编译器把多维数组作为嵌套数组的简写，在这种编译器下 $A[i]$ 和 $B[i]$ 一样也是一个一维数组。作者在这里显然不是指这种编译器。——译者注

5.3 数据结构中的函数

函数和具体的数据类型在数据结构中扮演了互补的角色。表和树构成了外部的框架并将信息组织起来，而函数则存储了可能的计算。虽然函数在计算机中表示为有限长的程序，但是我们却经常可以把它们看作是无穷对象。

序偶和表可以包括函数作为它们的分量：[⊖]

```
(concat, Math.sin);
> (fn, fn) : (string list -> string) * (real -> real)
[op+, op-, op*, op div, op mod, Int.max, Int.min];
> [fn, fn, fn, fn, fn] : (int * int -> int) list
```

在数据结构中存储的函数可以被取出来并进行应用。

```
val titlefns = [dukify, lordify, knightify];
> val titlefns = [fn, fn, fn] : (string -> string) list
hd titlefns "Gloucester";
> "The Duke of Gloucester" : string
```

这是一个柯里函数调用：*hd titlefns*返回了函数*dukify*。在这个例子中，多态函数*hd*具有类型

$(string \rightarrow string) list \rightarrow (string \rightarrow string)$

一棵包含函数的二叉搜索树可能会用于桌面计算器程序。计算器中的函数可以用名字来寻找。

```
val funtree = Dict.insert (Dict.insert (Dict.insert (Lf, "sin", Math.sin),
                                                "cos", Math.cos),
                           "atan", Math.atan);
> val funtree =
> Br (("sin", fn),
>     Br (("cos", fn), Br (("atan", fn), Lf, Lf), Lf),
>     Lf) : (real -> real) Dict.t
Dict.lookup (funtree, "cos") 0.0;
> 1.0 : real
```

176

在这棵树中存储的函数必须具有相同的类型，这里是 $real \rightarrow real$ 。虽然不同的类型也可以组合成一个数据类型，但是这会带来不便。就像在4.6节末所提到的，类型 exn 可以被认为包括所有的类型。上面函数的一个更为灵活的类型就是 $exn list \rightarrow exn$ 。

练习 5.5 在上面的例子中，多态函数*Dict.lookup*具有什么类型？

5.4 作为参数和结果的函数

第3章的排序函数是编写成对实数排序的。通过把比序谓词($<$)作为一个参数，这些函数可以被推广到任意的有序类型。下面是插入排序的多态函数：

```
fun insort lessequal =
  let fun ins (x, []) = [x]
      | ins (x, y::ys) =
          if lessequal(x,y) then x::y::ys
          else y :: ins (x,ys)
  in fun sort [] = []
  end
```

[⊖] 回忆一下关键字`op`将中缀操作符作为函数返回。

```

      | sort (x::xs) = ins (x, sort xs)
    in sort end;
> val insort = fn
> : ('a * 'a -> bool) -> 'a list -> 'a list

```

函数`ins`和`sort`是在局部定义的，它们引用了`lessequal`。虽然不是很明显，但`insort`确实是一个柯里函数。给定具有类型 $\tau \times \tau \rightarrow \text{bool}$ 的参数，它返回函数`sort`，具有类型 $\tau \text{ list} \rightarrow \tau \text{ list}$ 。用于比较的类型和表的元素类型必须一致。

现在整数也可以排序了。（虽然操作符`<=`是重载的，但是整数表约束了它的类型。）

```

insort (op<=) [5,3,7,5,9,8];
> [3, 5, 5, 7, 8, 9] : int list

```

将关系`>`作为`lessequal`传入则返回降序排序：

```

insort (op>=) [5,3,7,5,9,8];
> [9, 8, 7, 5, 5, 3] : int list

```

字符串序偶可以依照字典顺序排序：

```

fun leq_stringpair ((a,b), (c,d): string*string) =
  a<c orelse (a=c andalso b<=d);
> val leq_stringpair = fn
> : (string * string) * (string * string) -> bool

```

177 我们来排序（姓，名）序偶的表：

```

insort leq_stringpair
  [(("Herbert", "Walter"), ("Plantagenet", "Richard")),
   ("Plantagenet", "Edward"), ("Brandon", "William"),
   ("Tyrrell", "James"), ("Herbert", "John")];
> [(("Brandon", "William"), ("Herbert", "John")),
>  ("Herbert", "Walter"), ("Plantagenet", "Edward"),
>  ("Plantagenet", "Richard"), ("Tyrrell", "James")]
> : (string * string) list

```

函数在数值计算中经常被作为参数传递。下面的算子计算求和 $\sum_{i=0}^{m-1} f(i)$ 。为了效率起见，它使用了一个局部的迭代函数来引用`f`和`m`：

```

fun summation f m =
  let fun sum (i,z) : real =
        if i=m then z else sum (i+1, z+(f i))
      in sum(0, 0.0) end;
  > val summation = fn : (int -> real) -> int -> real

```

用`fn`记法可以很好地和算子配合。下面，在计算求和 $\sum_{k=0}^9 k^2$ 时，它省去了事先对平方函数的声明。

```

summation (fn k => real(k*k)) 10;
> 285.0 : real

```

二阶求和 $\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} g(i, j)$ 可以如下计算

```

summation (fn i => summation (fn j => g(i,j)) n) m;


```

这相当于 Σ -记法的ML翻译；索引变量 i 和 j 是通过 fn 绑定的。内求和 $\sum_{j=0}^{n-1} g(i, j)$ 是 i 的一个函数。作用在 j 上的函数则是 g 对 i 的函数部分应用。

上面的函数部分应用还可以简化，利用柯里函数 h 取代 g 来作为被加函数。二阶求和 $\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} h i j$ 可以如下计算

```
summation (fn i => summation (h i) n) m;
```

注意观察， $summation f$ 具有和 f 同样的类型，就是 $int \rightarrow real$ ，因此 $\sum_{i=0}^{m-1} \sum_{j=0}^{i-1} f(j)$ 可以通过 $summation (summation f) m$ 计算。

 多态 val 声明。因为函数可以是一个计算的结果，所以你可能会觉得下面的这些声明是合法的：

```
val list5 = replist 5;
> val list5 = fn : 'a -> 'a list
val f = hd [hd];
> val f = fn : 'a list -> 'a
```

178

在早期版本的ML中它们确实是合法的，但是现在会产生一个类似“多态声明中不为值的项”的信息。这个限制和引用有关，在8.3节有详细的解释。将函数声明由 $val f = E$ 改为

```
fun f x = E x
```

就可以使它合法了。这个改变不影响什么，除非计算 E 会产生副作用或开销很大。

这个限制会影响所有的多态 val 声明，而不仅仅是针对函数的。应该记得在交互环境顶层输入表达式 E 相当于输入 $val it = E$ 。因此，举例来说，不能输入 $hd [[]]$ 。

练习 5.6 书写一个多态函数来实现自顶向下的合并排序，将比序谓词（ $<$ ）作为参数传入。

练习 5.7 书写一个算子来计算关于函数 f 的最小值 $\min_{i=0}^{m-1} f(i)$ ，其中 m 是任意给定的正整数。用这个算子来表达二元最小值 $\min_{i=0}^{m-1} \min_{j=0}^{n-1} g(i, j)$ ， m 和 n 都是正整数。

通用算子

函数式程序员通常使用高阶函数来清楚简洁地表达程序。处理表的算子在Lisp的早期就已经非常普及了，而且有着无数的变体和各种名称。一些操作本来需要单独声明递归函数，而通过这些算子则可以直接表达。对于树也可以定义类似的递归算子。

一个全面的算子集相当于一种可以表达其他函数的抽象语言。在阅读本节之后，你会发现它有益于对前面的章节进行复习，并知道如何使用算子来简化那些函数的定义。

5.5 切片

想像一下将中缀操作符只应用在一个操作数上，或左或右，另一个操作数空缺。这定义了只有一个参数的函数，称为切片（section）。下面是使用Bird和Wadler（1988）记法的例子：

- ("Sir" ^)是函数*knightify*
- (/2.0)是函数“除以2”

179 切片可以通过算子*secl*和*secl*（相当粗糙地）加到ML中：

```
fun secl x f y = f(x,y);
> val secl = fn : 'a -> ('a * 'b -> 'c) -> 'b -> 'c
fun secr f y x = f(x,y);
> val secr = fn : ('a * 'b -> 'c) -> 'b -> 'a -> 'c
```

这两个算子通常和中缀函数以及*op*一起使用，不过也可以应用在任何具有合适类型的函数上。下面是几个左切片：

```
val knightify = (secl "Sir " op^);
> val knightify = fn : string -> string
knightify "Geoffrey";
> "Sir Geoffrey" : string
val recip = (secl 1.0 op/);
> val recip = fn : real -> real
recip 5.0;
> 0.2 : real
```

下面是一个右切片，表示除以2：

```
val halve = (secl op/ 2.0);
> val halve = fn : real -> real
halve 7.0;
> 3.5 : real
```

练习 5.8 切片和柯里函数之间有没有相似点？

练习 5.9 下面的切片产生的函数是什么？回忆一下*take*的作用是从表的前面提取元素（3.4节），而*inter*的作用是返回两个表的交集（3.15节）。

```
secl op@ ["Richard"]
secl ["heed", "of", "yonder", "dog!"] List.take
secl List.take 3
secl ["his", "venom", "tooth"] inter
```

5.6 组合子

λ -演算理论中的一部分与被称为组合子（combinator）的表达式有关。很多组合子可以在ML中编写成高阶函数，并且有实际的应用。

180 复合。中缀操作符*o*（没错，是字母“o”）表示了函数的复合。标准库中将它声明为：

```
infix o;
fun (f o g) x = f (g x);
> val o = fn : ('b -> 'c) * ('a -> 'b) -> 'a -> 'c
```

数学家对复合并不陌生；*f o g*是一个函数，它对参数首先应用*g*，然后应用*f*。复合可以表达很多函数，特别是利用切片。例如，函数

```
fn x => ~(Math.sqrt x)
fn a => "beginning" ^ a ^ "end"
fn x => 2.0 / (x-1.0)
```

都可以不使用它们的形式参数来表达:

```
~ o Math.sqrt
(secl "beginning" op^ ) o (secl op^ "end")
(secl 2.0 op/) o (secl op- 1.0)
```

为了计算求和 $\sum_{k=0}^9 \sqrt{k}$, 函数 `Math.sqrt` 和转换整数到实数的函数 `real` 被复合在一起了。复合比 `fn` 记法要易读:

```
summation (Math.sqrt o real) 10;
```

组合子 `S`、`K` 和 `I`。恒等组合子 `I`, 只是返回它的参数:

```
fun I x = x;
> val I = fn : 'a -> 'a
```

将函数与 `I` 复合没有任何效果:

```
knightify o I o (prefix "William ") o I;
> fn : string -> string
it "Catesby";
> "Sir William Catesby" : string
```

组合子 `K` 创建常函数。给定 `x`, 它创建了一个总是返回 `x` 的函数:

```
fun K x y = x;
> val K = fn : 'a -> 'b -> 'a
```

为了演示常函数, 让我们通过累加 $\sum_{i=0}^{m-1} z$ 来计算乘积 $m \times z$:

```
summation (K 7.0) 5;
> 35.0 : real
```

组合子 `S` 是函数复合的一般形式:

```
fun S x y z = x z (y z);
> val S = fn : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
```

λ -演算中的所有函数都可以仅用 `S` 和 `K` 来表达, 而且不使用任何变量! David Turner (1979) 曾经使用这一著名的事实实现了惰性求值: 因为不涉及任何变量, 所以不需要机制去绑定它们的值。事实上, 所有惰性函数式语言编译器都利用了这一技术的某种改进形式。

下面是一个著名的例子, 它展示了 `S` 和 `K` 的表达能力。恒等组合子 `I` 可以定义为 `S K K`:

```
S K K 17;
> 17 : int
```

练习 5.10 写出 `S K K 17` 的计算步骤。

练习 5.11 假设已知表达式 `E`, 里面包含中缀操作符、常量和变量, 其中有变量 `x` 且只出现了一次。叙述一种方法来表达函数 `fn x => E`, 使用 `I`, 切片和复合来代替 `fn` 记法。

5.7 表算子 `map` (映射) 和 `filter` (过滤)

算子 `map` 将一个函数应用到表的每个元素上, 返回由所有结果组成的表:

$$\text{map } f [x_1, \dots, x_n] = [f x_1, \dots, f x_n]$$

ML程序库是这样声明`map`的:

```
fun map f [] = []
  | map f (x::xs) = (f x) :: map f xs;
> val map = fn : ('a -> 'b) -> 'a list -> 'b list
map recip [0.1, 1.0, 5.0, 10.0];
> [10.0, 1.0, 0.2, 0.1] : real list
map size ["York", "Clarence", "Gloucester"];
> [4, 8, 10] : int list
```

算子`filter`将一个谓词（返回布尔值的函数）应用在表上，返回由输入表中所有满足该谓词的元素组成的表，元素的顺序不变。

```
fun filter pred [] = []
  | filter pred (x::xs) =
    if pred x then x :: filter pred xs
    else filter pred xs;
> val filter = fn : ('a -> bool) -> 'a list -> 'a list
filter (fn a => size a = 4)
  ["Hie", "thee", "to", "Hell", "thou", "cacodemon"];
> ["thee", "Hell", "thou"] : string list
```

182

柯里函数中的参数模式匹配和元组中的完全一样。上面两个算子都是柯里的：`map`接受类型为 $\sigma \rightarrow \tau$ 的函数作为参数，返回类型为 $\sigma \text{ list} \rightarrow \tau \text{ list}$ 的函数，而`filter`则接受类型为 $\tau \rightarrow \text{bool}$ 函数作为参数，返回类型为 $\tau \text{ list} \rightarrow \tau \text{ list}$ 的函数。

感谢柯里化，这些函数可以组合在一起应用到表的表上。可以看到，`map(map f) [l1, l2, ..., ln]`将`map f`应用到每一个表`l1, l2, ..., ln`上。

```
map (map double) [[1], [2,3], [4,5,6]];
> [[2], [4, 6], [8, 10, 12]] : int list list
map (map (implode o rev o explode))
  [{"When", "he", "shall", "split"},
   {"thy", "very", "heart", "with", "sorrow"}];
> [{"nehW", "eh", "llaHS", "tilps"},
   {"yht", "yrev", "traeh", "htiw", "worros"}]
> : string list list
```

类似地，`map(filter pred) [l1, l2, ..., ln]`将`filter pred`应用到每一个表`l1, l2, ..., ln`上。它返回了满足谓词`pred`的元素所组成的表的表。

```
map (filter (secc op< "m"))
  [{"my", "hair", "doth", "stand", "on", "end"},
   {"to", "hear", "her", "curses"}];
> [{"hair", "doth", "end"}, {"hear", "her", "curses"}]
> : string list list
```

很多表的函数可以用`map`和`filter`简单地编写出来。我们的矩阵转置函数（3.9节）可写成

```
fun transp ([]::_) = []
  | transp rows = map hd rows :: transp (map tl rows);
> val transp = fn : 'a list list -> 'a list list
transp [{"have", "done", "thy", "charm"},
        {"thou", "hateful", "withered", "hag!"}];
> [{"have", "thou"}, {"done", "hateful"},
```

```
> ["thy", "withered"], ["charm", "hag!"]]
> : string list list
```

回忆一下我们在3.15节是怎样利用成员关系来定义两个“集合”的交集的。就知道这个声明可以简化成一行：

```
fun inter(xs,ys) = filter (secl (op mem) ys) xs;
> val inter = fn : 'a list * 'a list -> 'a list
```

练习 5.12 说出怎样将形如

$$\text{map } f(\text{map } g \text{ } xs)$$

的表达式替换成只调用一次`map`的等价表达式。

183

练习 5.13 声明中缀操作符`andf`，使得

$$\text{filter } (\text{pred1 andf pred2}) \text{ } xs$$

返回和`filter pred1 (filter pred2 xs)`同样的值。

5.8 表算子`takewhile`和`dropwhile`

这两个算子利用一个谓词从表中截出一个初始段：

$$\underbrace{[x_0, \dots, x_{i-1}]}_{\text{takewhile}} \underbrace{[x_i, \dots, x_{n-1}]}_{\text{dropwhile}}$$

其中，元素都满足谓词的初始段，是由`takewhile`返回的：

```
fun takewhile pred [] = []
  | takewhile pred (x::xs) =
    if pred x then x :: takewhile pred xs
    else [];
> val takewhile = fn : ('a -> bool) -> 'a list -> 'a list
```

剩下的元素（如果有的话）从第一个不满足谓词的元素开始，这个表由`dropwhile`返回：

```
fun dropwhile pred [] = []
  | dropwhile pred (x::xs) =
    if pred x then dropwhile pred xs
    else x::xs;
> val dropwhile = fn : ('a -> bool) -> 'a list -> 'a list
```

这两个算子可以处理作为字符列表的文本。谓词`Char.isAlpha`可以识别一个字符是否为字母。倘若用这个谓词，`takewhile`可以从一个句子中返回第一个单词，而`dropwhile`则返回剩下的字符。

```
takewhile Char.isAlpha (explode "that deadly eye of thine");
> ["#\"t\"", "#\"h\"", "#\"a\"", "#\"t\""] : char list
dropwhile Char.isAlpha (explode "that deadly eye of thine");
> ["#\" \"", "#\"d\"", "#\"e\"", "#\"a\"", "#\"d\"", "#\"l\"", "..."] : char list
```

由于两个算子都是柯里函数，因此`takewhile`和`dropwhile`可以和其他算子组合。例如，`map (takewhile pred)`返回由初始段组成的表。

5.9 表算子`exists`（存在）和`all`（全称）

这两个算子报告表中是否有一些（或是全部）元素满足某个谓词。它们可以被看作是表

184 上的量词:

```
fun exists pred []      = false
  | exists pred (x::xs) = (pred x) orelse exists pred xs;
> val exists = fn : ('a -> bool) -> 'a list -> bool

fun all pred []         = true
  | all pred (x::xs)    = (pred x) andalso all pred xs;
> val all = fn : ('a -> bool) -> 'a list -> bool
```

通过柯里化, 这两个算子将一个作用在类型 τ 上的谓词转换成作用在类型 τ list上的谓词。成员测试函数 $x \text{ mem } xs$ 可以用一行表达:

```
fun x mem xs = exists (secl op= x) xs;
> val mem = fn : 'a * 'a list -> bool
```

函数 disjoint 测试了两个表是否不含相同的元素:

```
fun disjoint(xs,ys) = all (fn x => all (fn y => x<>y) ys) xs;
> val disjoint = fn : 'a list * 'a list -> bool
```

由于参数顺序的关系, exists 和 all 在嵌套时很难被当作量词理解, 因此很难看出 disjoint 是在测试“对于所有 x 属于 xs 和所有 y 属于 ys , $x \neq y$ ”。然而, exists 和 all 与其他算子结合得都很好。对于表的表, 实用的算子组合包括:

$\text{exists}(\text{exists } \text{pred})$
 $\text{filter}(\text{exists } \text{pred})$
 $\text{takewhile}(\text{all } \text{pred})$

5.10 表算子 foldl (左折叠) 和 foldr (右折叠)

这两个算子具有不同寻常的一般性。它们将有两个参数的函数应用在表的元素上:

$$\begin{aligned}\text{foldl } f \ e \ [x_1, \dots, x_n] &= f(x_n, \dots, f(x_1, e) \dots) \\ \text{foldr } f \ e \ [x_1, \dots, x_n] &= f(x_1, \dots, f(x_n, e) \dots)\end{aligned}$$

由于表达式是从内到外计算的, 因此 foldl 调用将 f 从左到右地应用在表的元素上, 而 foldr 调用则从右到左地将函数应用到表元素上。这两个算子声明为

```
fun foldl f e []      = e
  | foldl f e (x::xs) = foldl f (f(x, e)) xs;
> val foldl = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

fun foldr f e []      = e
  | foldr f e (x::xs) = f(x, foldr f e xs);
> val foldr = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

185

很多函数都可以用 foldl 和 foldr 来表达。表元素的求和可以通过从0开始反复地进行加法来计算:

```
val sum = foldl op+ 0;
> val sum = fn : int list -> int
sum [1,2,3,4];
> 10 : int
```

而乘积则通过从1开始反复进行乘法来计算，我们并不需要将函数事先绑定到一个标识符上：

```
foldl op* 1 [1,2,3,4];
> 24 : int
```

这些定义之所以正确，是因为0和1分别是+和×的单位元 (identity element)，换句话说，对于所有 k 有， $0 + k = k$ 以及 $1 \times k = k$ 。很多`foldl`和`foldr`的应用都是属于这个类型的。

这两个算子将类型为 $\sigma \times \tau \rightarrow \tau$ 的函数作为第一个参数。这个函数本身也可以用算子来表达。嵌套的`foldl`可以将表的表相加：

```
foldl (fn (ns,n) => foldl op+ n ns) 0 [[1], [2,3], [4,5,6]];
> 21 : int
```

这比`sum (map sum [[1], [2,3], [4,5,6]])`要更直接，后者需要构造一个和的中间表`[1, 5, 15]`。

表的构造函数 (操作符`::`) 正好具有所要求的类型。将它作为`foldl`的参数可以产生一个高效的翻转函数：

```
foldl op:: [] (explode "Richard");
> [#"d", #"r", #"a", #"h", #"c", #"i", #"R"] : char list
```

迭代式的长度计算也同样简单：

```
foldl (fn (_,n) => n+1) 0 (explode "Margaret");
> 8 : int
```

要将`xs`和`ys`连接起来，可以从`ys`开始，通过`foldr`来将`::`应用到`xs`的每一个元素上：

```
foldr op:: ["out", "thee?"] ["And", "leave"];
> ["And", "leave", "out", "thee?"] : string list
```

通过`foldr`应用连接操作可以将表的表连在一起，就像函数`List.concat`那样。注意，空表`[]`是连接操作的单位元：

```
foldr op@ [] [[1], [2,3], [4,5,6]];
> [1, 2, 3, 4, 5, 6] : int list
```

186

回忆一下，`newmem`向表中增加一个尚不在其内的元素 (3.15节)。通过`foldr`应用该函数可以构造元素互不相同的“集合”：

```
foldr newmem [] (explode "Margaret");
> [#"M", #"g", #"a", #"r", #"e", #"t"] : char list
```

通过应用基于`::`和`f`的函数可以表达`map f`：

```
fun map f = foldr (fn(x,l)=> f x :: l) [];
> val map = fn : ('a -> 'b) -> 'a list -> 'b list
```

两个`foldr`调用可以计算两个表的笛卡尔乘积：

```
fun cartprod (xs, ys) =
  foldr (fn (x, pairs) =>
    foldr (fn (y,l) => (x,y)::l) pairs ys)
    [] xs;
> val cartprod = fn : 'a list * 'b list -> ('a * 'b) list
```

使用`map`和`List.concat`可以更为清楚地计算笛卡尔乘积，代价是要建立一个中间表。首先需要

声明一个柯里的序偶函数:

```
fun pair x y = (x,y);
> val pair = fn : 'a -> 'b -> 'a * 'b
```

然后建立序偶表的表

```
map (fn a => map (pair a) ["Hastings","Stanley"])
  ["Lord","Lady"];
> [(["Lord", "Hastings"), ("Lord", "Stanley")],
>  [(["Lady", "Hastings"), ("Lady", "Stanley")]]
> : (string * string) list list
```

……最后把里面的序偶表连接起来形成笛卡尔乘积:

```
List.concat it;
> [(["Lord", "Hastings"), ("Lord", "Stanley"),
>  ("Lady", "Hastings"), ("Lady", "Stanley")]
> : (string * string) list
```

两个笛卡尔乘积的算法都可以推广,用其他关于 x 和 y 的函数替换 (x,y) ,这样就可以表示形如 $\{f(x,y) | x \in xs, y \in ys\}$ 的集合了。

❶ 算子和标准库。中缀操作符 \circ ,用于函数合成,是在顶层定义了的。在顶层还有表算子 map 、 $foldl$ 和 $foldr$,它们同时被另行声明为结构 $List$ 的组件,属于 $List$ 的还有 $filter$ 、 $exists$ 和 all 。结构 $ListPair$ 提供了 map 、 $exists$ 和 all 的变体,它们接受有两个参数的函数,并操作表的序偶。例如, $ListPair.map$ 将函数应用在两个表中相应元素所配成的序偶上:

187

$$ListPair.map\ f\ ([x_1, \dots, x_n], [y_1, \dots, y_n]) = [f(x_1, y_1), \dots, f(x_n, y_n)]$$

如果表的长度不一,多余的元素就被忽略掉了。同样的结果可以使用 $List.map$ 和 $ListPair.zip$ 来得到,但是这会构造一个中间表。

练习 5.14 利用算子来表达并集函数 $union$ (3.15节)。

练习 5.15 利用算子来简化矩阵乘法 (3.10节)。

练习 5.16 利用 $foldl$ 或 $foldr$ 来表达 $exists$ 。

练习 5.17 利用算子来表达集合表达式

$$\{x - y | x \in xs, y \in ys, y < x\}$$

5.11 更多递归算子的例子

二叉树和类似的递归数据类型可以利用递归算子来处理。甚至是自然数 $0, 1, 2, \dots$ 也可以看成是递归数据类型:它们的构造子是 0 和后继函数。

函数的幂。如果 f 是一个函数且 $n \geq 0$,那么 f^n 是一个函数,满足

$$f^n(x) = \underbrace{f(\dots f(f(x))\dots)}_{n\text{次}}$$

下面是函数 $repeat\ f\ n$:

```

fun repeat f n x =
  if n>0 then repeat f (n-1) (f x)
  else x;
> val repeat = fn : ('a -> 'a) -> int -> 'a -> 'a

```

令人惊讶的是，许多函数都具有这个形式。这方面的例子包括`drop`和`replist`（它们分别是在3.4节和5.2节声明的）：

```

repeat tl 5 (explode "I'll drown you in the malmsey-butt...");
> [# "d", # "r", # "o", # "w", # "n", # " ", ...] : char list
repeat (secl "Ha!" op::) 5 [];
> ["Ha!", "Ha!", "Ha!", "Ha!", "Ha!"] : string list

```

所有标签都是同一个常量的完全二叉树可以这样构造：

```

repeat (fn t=>Br("No",t,t)) 3 Lf;
> Br ("No", Br ("No", Br ("No", Lf, Lf),
>                      Br ("No", Lf, Lf)),
>      Br ("No", Br ("No", Lf, Lf),
>           Br ("No", Lf, Lf)))
> : string tree

```

188

重复一个合适的序偶函数可以计算阶乘：

```

fun factaux (k,p) = (k+1, k*p);
> val factaux = fn : int * int -> int * int
repeat factaux 5 (1,1);
> (6, 120) : int * int

```

树的递归。算子`treefold`，对于二叉树来说，类似于`foldr`。调用`foldr f e xs`，形象地说，是把表中的`::`替换成`f`以及把`nil`替换成`e`。给定一棵树，`treefold`将每个叶子替换成值`e`并把每个分支替换成对函数`f`的应用，`f`接受三个参数。

```

fun treefold f e Lf = e
  | treefold f e (Br(u,t1,t2)) = f(u, treefold f e t1, treefold f e t2);
> val treefold = fn
> : ('a * 'b * 'b -> 'b) -> 'b -> 'a tree -> 'b

```

这个算子可以表示上一章很多关于树的函数。函数`size`将每一个叶子都替换成0并将每一个分支替换成一个将1加到子树大小上去的函数：

```

treefold (fn (_,c1,c2) => 1+c1+c2) 0

```

函数`depth`在每一个分支处取最大值：

```

treefold (fn (_,d1,d2) => 1 + Int.max(d1,d2)) 0

```

调换`Br`的两个子树的函数，通过树的递归，就定义了`reflect`（镜像）：

```

treefold (fn (u,t1,t2) => Br(u,t2,t1)) Lf

```

要计算前序表，就在每个分支处将标签和对应于子树的表连接起来：

```

treefold (fn (u,l1,l2) => [u] @ l1 @ l2) []

```

关于项的操作。项（term）的集合 $x, f(x), g(x, f(x)), \dots$ ，是通过变量和函数应用来产生的，这与ML的数据类型是相对应的

```
datatype term = Var of string
              | Fun of string * term list;
```

项 $(x + u) - (y \times x)$ 可以声明为

```
val tm = Fun("-", [Fun("+", [Var "x", Var "u"]),
                  Fun("×", [Var "y", Var "x"])]);
```

189

虽然用ML的表来表示函数的参数是非常自然的，但是类型`term`和`term list`必须被看作是相互递归的。典型的作用于项上的函数都需要使用与之配合的作用在项列表上的函数。幸运的是，这个配合函数不需要另外定义；大多数情况下它都可以利用表算子来表达。

如果有ML函数 $f: \text{string} \rightarrow \text{term}$ 定义了一个从变量到项的替换，那么下面的`subst f`可以把这个替换扩展到整个项中的所有变量上去。观察一下`map`是怎样将替换应用到项列表上去的。

```
fun subst f (Var a)      = f a
  | subst f (Fun(a,args)) = Fun(a, map (subst f) args);
> val subst = fn : (string -> term) -> term -> term
```

项中所有变量的列表也可以通过`map`来计算：

```
fun vars (Var a)      = [a]
  | vars (Fun(_,args)) = List.concat (map vars args);
> val vars = fn : term -> string list
vars tm;
> ["x", "u", "y", "x"] : string list
```

这个办法有点浪费，因为`List.concat`不断地将表来回复制。换个办法，声明函数`accumVars`，利用一个额外的参数来积累变量表。这个函数要通过`foldr`来扩展到项列表上（积累需要两个参数）：

```
fun accumVars (Var a, bs)      = a::bs
  | accumVars (Fun(_,args), bs) = foldr accumVars bs args;
> val accumVars = fn : term * string list -> string list
accumVars (tm, []);
> ["x", "u", "y", "x"] : string list
```

下面是替换的演示。一个简单的替换函数`replace t a`将名为`a`的变量替换成项`t`，其他名字的变量则不变：

```
fun replace t a b = if a=b then t else Var b;
> val replace = fn : term -> string -> string -> term
```

因此，`subst (replace t a) u`将项`u`里面所有名为`a`的变量都替换成了项`t`。用`-z`替换掉`tm`里的`x`则产生了项 $(-z + u) - (y \times -z)$ ：

```
subst (replace (Fun("-", [Var "z"])) "x") tm;
> Fun("-",
>   [Fun("+", [Fun("-", [Var "z"]), Var "u"]),
>   Fun("×", [Var "y", Fun("-", [Var "z"]])])])
> : term
```

现在变量表里面`z`替代了原来`x`的位置：

```
accumVars (it, []);
> ["z", "u", "y", "z"] : string list
```

190

练习 5.18 声明算子 *prefold*, 使得 *prefold f e t* 等价于 *foldr f e (preorder t)*。

练习 5.19 书写函数 *nf*, 使得 *repeat nf* 可以计算斐波那契数。

练习 5.20 下面的函数有什么用处?

```
fun funny f 0 = 1
  | funny f n = if n mod 2 = 0
                then funny (f o f) (n div 2)
                else funny (f o f) (n div 2) o f;
```

练习 5.21 *treefold F I* 是什么函数? 其中 *F* 的定义如下:

```
fun F (v,f1,f2) vs = v :: f1 (f2 vs);
```

练习 5.22 考虑对项中的 *Fun* 结点进行计数。首先使用 *vars* 模式来表达这个函数, 然后使用 *accumVars* 模式, 最后不要使用算子来表达这个函数。

练习 5.23 注意 *vars tm* 的结果中出现了两次 *x*。书写一个函数来计算一项中没有重复的变量表。利用算子, 你能否发现一个简单的办法呢?

序列, 或无穷表

惰性表是函数式程序设计最著名的特性之一。惰性表中的元素直到程序的其他部分需要时才会被计算; 因此惰性表可以是无穷的。在惰性求值的语言如 Haskell 中, 所有的数据结构都是惰性求值的, 并且无穷表在程序中很常见。在 ML 中, 计算是严格的, 无穷表就很少见了。这一节叙述了在 ML 中怎样表达无穷表, 通过一个函数表示表尾来延时对它的求值。

认识用无穷表进行程序设计的害处也是很重要的。迄今, 我们都是希望每个函数, 从最大公因子到优先队列, 都能在有限时间内得到结果。递归被用于将问题简化成更简单的子问题。每个递归函数都包括一个基本情形, 在那儿函数可以停下来。

现在需要处理潜在的无穷结果。我们可以看到无穷表的任何有穷部分, 但想看到全部是不可能的。可以通过依次将两个无穷表的元素相加来取得和的表, 但是不能翻转无穷表或找到其中最小的元素。这里将要定义无穷进行而没有基本情形的递归函数。我们不再询问程序能否停止运行, 而只能问程序的每个有穷部分能否在有限时间内产生。

ML 中作用在无穷表上的函数要比在惰性求值语言中的相应函数复杂。然而, 通过将运行机制暴露出来, 可以帮助我们避免一些缺陷。从运行机制方面来思考应该不是唯一的办法, 而在无穷值上进行的计算可能会超出我们的想像力。论域理论 (domain theory) 给出了关于这种计算的更为深入的观点 (Gunter, 1992; Winskel, 1993)。

191

5.12 序列类型

无穷表传统上称为流 (stream), 不过这里我们将其称为序列 (sequence)。(“流”在 ML 中指一个输入输出通道)。和表一样, 序列或者是空的, 或者是包含头和尾的。空序列是 *Nil*, 而非空序列具有形式 *Cons(x, xf)*, 其中 *x* 是头, *xf* 是计算尾的函数: [⊖]

⊖ 单元类型 *unit* 在 2.8 节有叙述。

```
datatype 'a seq = Nil
                | Cons of 'a * (unit -> 'a seq);
```

从这个声明开始,我们将通过和表进行类比起来交互地开发一套序列的基本操作。稍后,为了避免名字冲突,将把它们组织到一个恰当的结构中去。

返回头和尾的函数很容易声明。和表一样,检查到空序列时应该抛出一个异常:

```
exception Empty;
fun hd (Cons(x,xf)) = x
  | hd Nil          = raise Empty;
> val hd = fn : 'a seq -> 'a
```

为了检查序列尾,将函数 xf 应用到 $()$ 上。这个参数是单元类型 $unit$ 唯一的值,不传递任何信息,它仅仅是迫使对序列尾进行求值。

```
fun tl (Cons(x,xf)) = xf()
  | tl Nil          = raise Empty;
> val tl = fn : 'a seq -> 'a seq
```

调用 $cons(x, xq)$ 将头 x 和尾序列 xq 合并到一起组成一个更长的序列:

```
fun cons(x,xq) = Cons(x, fn()=>xq);
> val cons = fn : 'a * 'a seq -> 'a seq
```

注意, $cons(x, E)$ 并不是惰性求值的。ML对表达式 E 求值,得到结果 xq ,并返回 $Cons(x, fn () => xq)$ 。所以在 $cons$ 里面的 fn 并没有对序列尾的求值进行延时。 $cons$ 只能用在不需要惰性求值的地方,例如将一个表转换成序列:

```
fun fromList l = List.foldr cons Nil l;
> val fromList = fn : 'a list -> 'a seq
```

要想延迟对 E 求值,直接写 $Cons(x, fn () => E)$ 代替 $cons(x, E)$ 。现在来定义从 k 开始递增的整数序列:

```
fun from k = Cons(k, fn()=> from(k+1));
> val from = fn : int -> int seq
from 1;
> Cons (1, fn) : int seq
```

这个序列从1开始,下面是另外几个例子:

```
tl it;
> Cons (2, fn) : int seq
tl it;
> Cons (3, fn) : int seq
```

调用 $take(xq, n)$ 取得序列 xq 的前 n 个元素作为表返回:

```
fun take (xq, 0)      = []
  | take (Nil, n)     = raise Subscript
  | take (Cons(x,xf), n) = x :: take (xf(), n-1);
> val take = fn : 'a seq * int -> 'a list
take (from 30, 7);
> [30, 31, 32, 33, 34, 35, 36] : int list
```

计算是怎样进行的呢? 计算 $take(from\ 30, 2)$ 是这样进行的:

```

take(from 30, 2)
⇒ take(Cons(30, fn() => from(30 + 1)), 2)
⇒ 30 :: take(from(30 + 1), 1)
⇒ 30 :: take(Cons(31, fn() => from(31 + 1)), 1)
⇒ 30 :: 31 :: take(from(31 + 1), 0)
⇒ 30 :: 31 :: take(Cons(32, fn() => from(32 + 1)), 0)
⇒ 30 :: 31 :: []
⇒ [30, 31]

```

可以看到元素32被计算了，但却没有用到。类型 α seq并不是真正延时，非空序列的头总是会被预先计算。更糟的是，重复地检查序列尾导致对其重复计算，我们没有传需调用，只有传名调用。通过增加额外的复杂性作为代价，这些缺陷是可以改进的（8.4节）。

193

练习 5.24 解释下面版本的`from`错在哪里，叙述`take(badfrom 30, 2)`的计算步骤。

```
fun badfrom k = cons(k, badfrom(k+1));
```

练习 5.25 下面的 α seq变种将所有非空序列表示为一个函数，防止了过早地对首元素进行计算（Reade, 1989, 第324页）。为这种类型的序列编写函数`from`和`take`:

```

datatype 'a seq = Nil
                | Cons of unit -> 'a * 'a seq;

```

练习 5.26 下面的 α seq变种，通过相互递归进行声明，比上面的更具惰性。所有序列都是函数，甚至判断序列是否为空所需的计算都被延时了。为这种类型的序列编写函数`from`和`take`:

```

datatype 'a seqnode = Nil
                  | Cons of 'a * 'a seq
and 'a seq = Seq of unit -> 'a seqnode;

```

5.13 基本的序列处理

为了使基于序列的函数可计算，输出的每个有穷部分都必须仅依赖于输入的某一有穷部分。考虑将序列里的整数依次地进行平方。在对输出序列的尾部进行求值时，需要将`squares`应用于输入序列的尾部。

```

fun squares Nil : int seq = Nil
  | squares (Cons(x, xf)) = Cons(x*x, fn() => squares(xf()));
> val squares = fn : int seq -> int seq
squares (from 1);
> Cons (1, fn) : int seq
take (it, 10);
> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100] : int list

```

将两个序列的对应元素相加也是类似的。对输出的尾部求值时会去对两个输入的尾部求值。只要有一个输入序列为空时，输出就为空。

```

fun add (Cons(x, xf), Cons(y, yf)) = Cons(x+y,
                                           fn() => add(xf(), yf()))
  | add _ : int seq = Nil;

```

194


```
> val add = fn : int seq * int seq -> int seq
add (from 10000, squares (from 1));
> Cons (10001, fn) : int seq
take (it, 5);
> [10001, 10005, 10011, 10019, 10029] : int list
```

序列的追加函数和表的类似。 $xq @ yq$ 的元素首先从 xq 中提取；当 xq 为空时，就改为从 yq 中提取。

```
fun Nil @ yq = yq
  | (Cons(x,xf)) @ yq = Cons(x, fn()=> (xf()) @ yq);
> val @ = fn : 'a seq * 'a seq -> 'a seq
```

为了进行一个简单的演示，让我们用`fromList`建立一个有穷序列。

```
val finite = fromList [25,10];
> Cons (25, fn) : int seq
finite @ from 1415;
> Cons (25, fn) : int seq
take(it, 3);
> [25, 10, 1415] : int list
```

如果 xq 是无穷的，那么 $xq @ yq$ 就等于 xq 。追加函数的一个变种可将两个无穷序列均匀地合在一起。两个序列的元素可以交替（interleaved）存放：

```
fun interleave (Nil, yq) = yq
  | interleave (Cons(x,xf), yq) =
    Cons(x, fn()=> interleave(yq, xf()));
> val interleave = fn : 'a seq * 'a seq -> 'a seq
take(interleave(from 0, from 50), 10);
> [0, 50, 1, 51, 2, 52, 3, 53, 4, 54] : int list
```

在递归调用中，`interleave`将两个序列进行交换，使得任何一个都不能把另一个排除在外。

序列的算子。像`map`和`filter`这样的表算子也可以推广到序列上。函数`squares`就是算子`map`的一个例子，它将一个函数应用到序列的每一个元素上：

```
fun map f Nil = Nil
  | map f (Cons(x,xf)) = Cons(f x, fn()=> map f (xf()));
> val map = fn : ('a -> 'b) -> 'a seq -> 'b seq
```

为了过滤一个序列，需要连续调用尾函数直到发现满足给定谓词的元素为止。如果没有这样的元素，计算就不会停止。

```
fun filter pred Nil = Nil
  | filter pred (Cons(x,xf)) =
    if pred x then Cons(x, fn()=> filter pred (xf()))
    else filter pred (xf());
> val filter = fn : ('a -> bool) -> 'a seq -> 'a seq
filter (fn n => n mod 10 = 7) (from 50);
> Cons (57, fn) : int seq
take(it, 8);
> [57, 67, 77, 87, 97, 107, 117, 127] : int list
```

函数`from`是算子`iterates`的一个例子，它生成一个形如 $[x, f(x), f(f(x)), \dots, f^k(x), \dots]$ 的序列：

```

fun iterates f x = Cons(x, fn()=> iterates f (f x));
> val iterates = fn : ('a -> 'a) -> 'a -> 'a seq
iterates(secr op/ 2.0) 1.0;
> Cons (1.0, fn) : real seq
take(it, 5);
> [1.0, 0.5, 0.25, 0.125, 0.0625] : real list

```

序列结构。让我们再把刚才探讨过的几个函数凑在一起做成一个结构。就像在二叉树结构中一样(4.13节)，把数据类型声明留在外面，这样可以直接引用构造子。现在假设其他的序列基本函数没有在顶层声明，而是在结构Seq中声明的，它满足下面的签名：

```

signature SEQUENCE =
  sig
    exception Empty
    val cons      : 'a * 'a seq -> 'a seq
    val null      : 'a seq -> bool
    val hd        : 'a seq -> 'a
    val tl        : 'a seq -> 'a seq
    val fromList  : 'a list -> 'a seq
    val toList    : 'a seq -> 'a list
    val take      : 'a seq * int -> 'a list
    val drop      : 'a seq * int -> 'a seq
    val @         : 'a seq * 'a seq -> 'a seq
    val interleave : 'a seq * 'a seq -> 'a seq
    val map       : ('a -> 'b) -> 'a seq -> 'b seq
    val filter    : ('a -> bool) -> 'a seq -> 'a seq
    val iterates  : ('a -> 'a) -> 'a -> 'a seq
    val from      : int -> int seq
  end;

```

196

练习 5.27 声明尚缺的函数`null`和`drop`，类比表的相应函数。同时声明`toList`来将有穷序列转换成表。

练习 5.28 说明`add(from 5, squares(from 9))`的计算步骤。

练习 5.29 声明一个函数，对于给定的正整数 k ，将序列 $[x_1, x_2, \dots]$ 转换成一个新的序列，其中原序列的每个元素在新序列中重复 k 次：

$$[\underbrace{x_1, \dots, x_1}_{k \text{ 次}}, \underbrace{x_2, \dots, x_2}_{k \text{ 次}}, \dots]$$

练习 5.30 声明一个函数来将序列中相邻的元素相加，即将序列 $[x_1, x_2, x_3, x_4, \dots]$ 转换成序列 $[x_1 + x_2, x_3 + x_4, \dots]$ 。

练习 5.31 在表算子`takewhile`、`dropwhile`、`exists`和`all`中，哪一个或哪几个可以合理地推广到无穷序列中去？编写那些可以推广的，并解释那些不能推广的算子有何不妥。

5.14 基本的序列应用

我们可以利用Seq结构来找零钱、表示随机数的无穷序列以及枚举素数。下面举几个例子来重点说明序列算子。

再议找零钱问题。函数`allChange` (3.7节) 算出了所有可能的找零钱方法。这很不实用：如果是用英国硬币的话，一共有4366种方法来找99便士！

如果函数返回一个序列，那么它可以在需要时计算解法，这节省了时间和空间。要想在ML中得到希望的效果就需要注意了。仅仅是将`allChange`中的表操作替换成序列操作几乎没什么用。新函数会包含两个递归调用，没有任何东西可以延迟第二个调用的执行，因此结果序列还是会被完全计算出来。

```
Seq.@ (allChange(c::coins, c::coinvals, amount-c),
      allChange(coins, coinvals, amount))
```

更好的办法是从练习3.14的答案开始，其中追加被取代为使用一个参数去积累找零钱的解法。

197

这个作为累加器的参数通常是一个表。我们是不是应该把它换成序列呢？

```
fun seqChange (coins, coinvals, 0, coinsf) = Cons(coins, coinsf)
  | seqChange (coins, [], amount, coinsf) = coinsf()
  | seqChange (coins, c::coinvals, amount, coinsf) =
    if amount < 0 then coinsf()
    else seqChange(c::coins, c::coinvals, amount-c,
                  fn() => seqChange(coins, coinvals, amount, coinsf));
> val seqChange = fn : int list * int list * int *
> (unit -> int list seq) -> int list seq
```

这里并没有直接使用序列，而是用了一个尾函数`coinsf`，具有类型`unit -> int list seq`。这可以让我们在第一行使用`Cons`，而不是急于计算其参数的`Seq.cons`。并且还需要在内层递归中使用`fn`来将它延时。这类处理在Haskell中要容易些。

现在我们来枚举解，每一个都可立即得到：

```
seqChange([], gb.coins, 99, fn() => Nil);
> Cons ([2, 2, 5, 20, 20, 50], fn) : int list seq
Seq.tl it;
> Cons ([1, 1, 2, 5, 20, 20, 50], fn) : int list seq
Seq.tl it;
> Cons ([1, 1, 1, 1, 5, 20, 20, 50], fn) : int list seq
```

增加的开销是不大的。计算所有的解需要354毫秒，比这个函数使用表的版本要慢大约1/3，但却是原来`allChange`的两倍速度。

随机数。在3.18节，我们为排序的例子生成了由10 000个随机数组成的表。然而，通常都不能事先知道需要多少个随机数。一般来说，随机数发生器是一个过程，它在局部变量中存储了一个随机数“种子”。在函数式语言中，可以定义一个无穷的随机数序列。这个做法隐藏了实现的细节，并且可以在需要时生成随机数。

```
local val a = 16807.0 and m = 2147483647.0
  fun nextRand seed =
    let val t = a*seed
    in t - m * real(floor(t/m)) end
in
  fun randseq s = Seq.map (secur op/ m)
    (Seq.iterates nextRand (real s))
end;
> val randseq = fn : int -> real seq
```

观察一下`Seq.iterates`是怎样生成一个数的序列的，而其中每一个元素都通过`Seq.map`被除以`m`。最后的随机数是在0和1之间的实数，但不包括0和1。使用`Seq.map`可以将它们转换成0到9之间

的整数:

198

```
Seq.map (floor o secl(10.0) op* ) (randseq 1);
> Cons (0, fn) : int seq
Seq.take (it, 12);
> [0, 0, 1, 7, 4, 5, 2, 0, 6, 6, 9, 3] : int list
```

素数。素数的序列可以用埃拉托色尼筛法 (Sieve of Eratosthenes) 来计算。

- 从序列[2, 3, 4, 5, 6, ...]开始。
- 把2作为素数。删除其他所有2的倍数, 因为它们不可能是素数。这使得序列剩下[3, 5, 7, 9, 11, ...]。
- 把3作为素数, 并删除它的其他倍数。这使得序列剩下[5, 7, 11, 13, 17, ...]。
- 把5作为素数……

在每一步中, 序列都只包含那些不能被已经生成的素数整除的数。因此序列的首元素是素数, 并且这一过程可以无限地进行下去。

函数sift删除序列中的某个数的倍数, 而sieve不断地对序列进行筛选 (sift)。

```
fun sift p = Seq.filter (fn n => n mod p <> 0);
> val sift = fn : int -> int seq -> int seq
fun sieve (Cons(p,nf)) = Cons(p, fn()=> sieve (sift p (nf())));
> val sieve = fn : int seq -> int seq
```

素数序列primes是从sieve [2, 3, 4, 5, ...]得出的。在序列被检查之前, 除首元素外, 并不会生成其他素数。

```
val primes = sieve (Seq.from 2);
> val primes = Cons (2, fn) : int seq
Seq.take (primes, 25);
> [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
> 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97] : int list
```

当我们书写这样的程序时, ML的类型可以帮助我们防止序列和尾函数之间的混淆。序列具有类型 $\tau \text{ seq}$, 而尾函数具有类型 $\text{unit} \rightarrow \tau \text{ seq}$ 。我们可以通过加入函数调用 $\dots()$ 或函数抽象 $\text{fn}() \Rightarrow \dots$ 来纠正类型错误。^①

5.15 数值计算

序列在数值分析中也有应用。初看上去这有点意外, 不过, 毕竟很多数值方法都是基于无穷数列的。为什么不直接表达它们呢?

199

平方根是一个简单的例子。回忆一下牛顿-拉夫森方法, 它用来计算某个数 a 的平方根。从一个正数近似值 x_0 开始, 通过下面的公式计算下一个近似值

$$x_{k+1} = \left(\frac{a}{x_k} + x_k \right) / 2$$

当相邻的两个近似值足够接近时, 计算就可以停止了。使用序列, 可以直接进行这个计算。

① 埃拉托色尼筛法几乎是所有函数式语言课本里面的例子, 特别是采用惰性求值的语言。它初步展示了函数式语言中无穷数据结构简洁和令人吃惊的表达能力。——译者注

函数`nextApprox`根据 x_k 计算 x_{k+1} 。将这个函数进行迭代则计算出一系列的近似值。

```
fun nextApprox a x = (a/x + x) / 2.0;
> val nextApprox = fn : real -> real -> real
Seq.take(Seq.iterates (nextApprox 9.0) 1.0, 7);
> [1.0, 5.0, 3.4, 3.023529412, 3.000091554,
> 3.000000001, 3.0] : real list
```

最简单的停止测试条件是当两个近似值差的绝对值小于某个给定的允许误差 $\varepsilon > 0$ (下面写作`eps`)。^①

```
fun within (eps:real) (Cons(x,xf)) =
  let val Cons(y,yf) = xf()
  in if Real.abs(x-y) < eps then y
    else within eps (Cons(y,yf))
  end;
> val within = fn : real -> real seq -> real
```

让 10^{-6} 作为允许误差, 1作为初始近似值, 则得到了一个平方根函数:

```
fun qroot a = within 1E~6 (Seq.iterates (nextApprox a) 1.0);
> val qroot = fn : real -> real
qroot 5.0;
> 2.236067977 : real
it*it;
> 5.0 : real
```

用Fortran来编写这个程序不会更好吗? 这个例子来自Hughes (1989) 以及Halfant和Sussman (1988), 他们说明了涉及序列的那些可独立互换的部分是怎样组合到数值算法中去的。每个算法都是为适合它的应用而定制的。

例如, 有很多种终止测试的方法可以选用。通过`within`来测试绝对差 ($|x-y| < \varepsilon$) 的办法对于大数来说就过于严格了。我们可以测试相对差 ($|x/y-1| < \varepsilon$) 或更有意思的:

$$\frac{|x-y|}{(|x|+|y|)/2+1} < \varepsilon$$

有时甚至应该更为谨慎地测试三个或更多的近似值是否足够接近。

每个终止测试都可以包装成一个从序列到实数的函数。像Richardson插补法这样的技术 (用于加速级数的收敛) 可以包装成从序列到序列的函数。这些函数可以组合起来进行数值微分和积分等计算。

练习 5.32 通过生成无穷和序列来计算

$$e^x = \frac{1}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^k}{k!} + \cdots$$

指数函数 e^x 。

练习 5.33 书写一个ML函数, 使用上面提到的另外两种终止测试来从序列中取得一个值。利

① 这里的递归调用传递的是`Cons(y, yf)`, 而不是调用具有同样值的`xf()`, 以避免调用两次`xf()`。应该记得我们的序列不是真正的惰性求值, 只是使用了传名调用规则。

用它声明平方根（或指数）函数。

5.16 交替和序列的序列

给定无穷序列 xq 和 yq ，考虑形成一个所有序偶 (x, y) 的序列，其中 x 来自 xq 且 y 来自 yq 。这个问题描绘了无穷数据结构计算的微妙之处。

就像在上面5.10节所提到的，可以通过 map 使用柯里的序偶函数 $pair$ 来生成一个表的表。类似地也可以生成一个序列的序列：

```
fun makeqq (xq,yq) = Seq.map (fn x=> Seq.map (pair x) yq) xq;
> val makeqq = fn : 'a seq * 'b seq -> ('a * 'b) seq seq
```

序列的序列可以通过 $takeqq(xqq, (m, n))$ 来查看。返回的表的表是 xqq 左上角 $m \times n$ 的矩形。

201

```
fun takeqq (xqq, (m,n)) = map (secc Seq.take n) (Seq.take(xqq,m));
> val takeqq = fn
> : 'a seq seq * (int * int) -> 'a list list
makeqq (Seq.from 30, primes);
> Cons (Cons ((30, 2), fn), fn) : (int * int) seq seq
takeqq (it, (3,5));
> [[(30, 2), (30, 3), (30, 5), (30, 7), (30, 11)],
> [(31, 2), (31, 3), (31, 5), (31, 7), (31, 11)],
> [(32, 2), (32, 3), (32, 5), (32, 7), (32, 11)]]
> : (int * int) list list
```

函数 $List.concat$ 将表的元素追加在一起，形成一个表。让我们也声明一个类似的枚举函数 $enumerate$ 来合并序列的序列。因为序列可能是无穷的，所以必须使用交替函数 $interleave$ 来取代追加函数。

下面是主要思想。如果输入的序列具有头 xq 和尾 xqq ，那么就递归地枚举 xqq 并将结果和 xq 进行交替。然而如果我们以 $List.concat$ 作为模型，那么将会得到一个错误的代码：

```
fun enumerate Nil = Nil
  | enumerate (Cons(xq,xqf)) = Seq.interleave(xq, enumerate (xqf()));
> val enumerate = fn : 'a seq seq -> 'a seq
```

如果这个函数的输入是无穷的，ML将产生一系列无限的递归调用，而不会输出任何结果。这个版本在惰性求值的函数式语言中也许是对的，但是在ML中当有输出产生时，必须显式地终止递归。这需要更复杂的情形分析。如果输入的序列非空，那么要继续分析它的首元素序列；如果仍是非空，那么里面就包含了一个可以输出的元素。

```
fun enumerate Nil = Nil
  | enumerate (Cons(Nil, xqf)) = enumerate (xqf())
  | enumerate (Cons(Cons(x,xf), xqf)) =
    Cons(x, fn()=> Seq.interleave(enumerate (xqf()), xf()));
> val enumerate = fn : 'a seq seq -> 'a seq
```

上面第二种和第三种情形很像错误版本中对交替函数 $interleave$ 的使用，不过内部的 $fn()=> \dots$ 终止了递归调用。

下面是所有正整数序偶的序列。

```
val pairqq = makeqq (Seq.from 1, Seq.from 1);
> val pairqq = Cons (Cons ((1, 1), fn), fn)
```

```

> : (int * int) seq seq
Seq.take(enumerate pairqq, 18);
> [(1, 1), (2, 1), (1, 2), (3, 1), (1, 3), (2, 2), (1, 4),
>  (4, 1), (1, 5), (2, 3), (1, 6), (3, 2), (1, 7), (2, 4),
>  (1, 8), (5, 1), (1, 9), (2, 5)] : (int * int) list

```

202 我们可以更为精确地描述枚举的顺序。看看下面的声明:

```

fun powof2 n = repeat double n 1;
> val powof2 = fn : int -> int
fun pack(i,j) = powof2(i-1) * (2*j - 1);
> val pack = fn : int * int -> int

```

这个压缩函数, $pack(i, j) = 2^{i-1}(2j - 1)$, 建立了正整数和正整数序偶间一一对应的关系。因此, 两个可数集的笛卡尔乘积仍是可数集。下面是这个函数值的一个小表格:

```

val nqq = Seq.map (Seq.map pack) pairqq;
> val nqq = Cons (Cons (1, fn), fn) : int seq seq
takeqq (nqq, (4,6));
> [[1, 3, 5, 7, 9, 11],
>  [2, 6, 10, 14, 18, 22],
>  [4, 12, 20, 28, 36, 44],
>  [8, 24, 40, 56, 72, 88]] : int list list

```

我们用枚举函数将压缩函数解码, 返回一个按自然顺序排列的正整数序列:

```

Seq.take (enumerate nqq, 12);
> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] : int list

```

不难看出为什么是这样的。每次交替都从一个序列中取一半元素, 而从另一个序列中取另一半。不断交替地将元素在输出序列中的位置按照二的幂来分配, 就像在压缩函数中的一样。

练习 5.34 预测, 或至少解释, ML对于下面表达式的回应:

```
enumerate (Seq.iterates I Nil);
```

练习 5.35 生成一个所有有限长正整数表的序列。(提示: 首先声明一个函数来生成给定长度正整数表的序列。)

练习 5.36 证明对于所有正整数 k , 存在唯一一对正整数 i 和 j 使得 $k = pack(i, j)$ 。 $pack(i, j)$ 的二进制表示是什么?

练习 5.37 通过改编类型 αseq 的定义来声明一个无穷二叉树类型。书写函数 itr , 当应用到整数 n 上时, 构造一棵无穷二叉树, 其根结点的标签是 n , 两棵子树分别为 $itr(2n)$ 和 $itr(2n + 1)$ 。

练习 5.38 (接上题。)书写函数来构造一个由给定的无穷二叉树的所有标签组成的序列。其中, 标签的枚举顺序是什么? 然后书写一个逆函数, 构造一棵无穷二叉树, 其标签来自一个无穷序列。

203

搜索策略和无穷表

定理证明、计划制定以及其他人工智能的应用都需要搜索。有很多种搜索策略:

- 深度优先搜索开销很小, 不过它可能会走一条死胡同, 永远走下去而找不到任何解。

- 广度优先搜索是完全的 (complete)，肯定可以找到所有的解，但是它需要大量的空间。
- 深度优先迭代深化也是完全的，只需要很少的空间，但是可能很慢。
- 最佳优先搜索必须由一个估计解距离的函数来引导。

通过将解的集合表示为一个惰性表，搜索策略的选择可以独立于对解的提取和使用。惰性表扮演了通讯通道的角色：生产者生成它的元素，同时，消费者将元素移走。由于表是惰性的，因此它的元素在消费者需要之前是不会生成的。

图5-1和图5-2将深度优先策略和广度优先策略进行了对比，将两者应用到同一棵树上。图中所描绘的是搜索过程中某一时刻的树，尚未访问到的子树被画成楔形。在本节中，所有的树的分支数都是有限的，但树的深度可能是无限的。

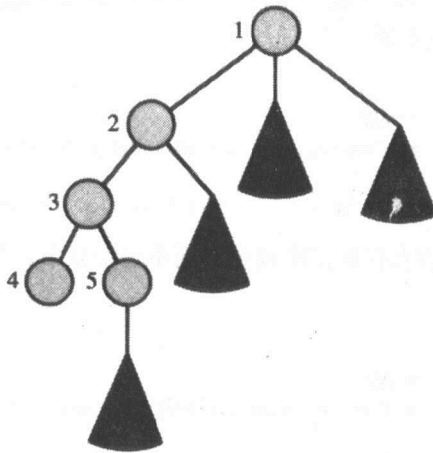


图5-1 深度优先搜索树

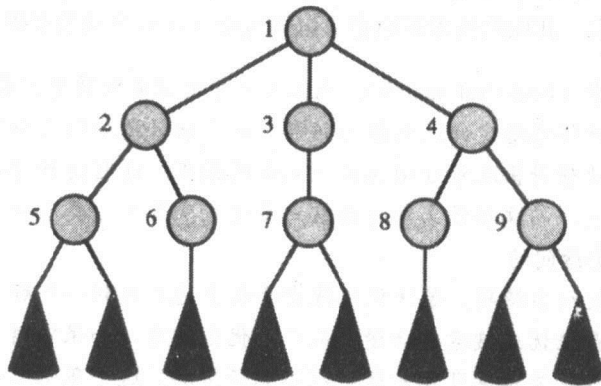


图5-2 广度优先搜索树

在深度优先搜索 (depth-first search) 中，先完全搜索每棵子树，然后才轮到它右边的兄弟。图中的数字显示了搜索的顺序。因为结点4是个叶子，所以才会轮到访问结点5，同时还有四棵子树等待访问。如果在结点5下面的子树是无穷的，那么就永远不会到达其他子树了：这个策略是不完全的。深度优先搜索通常被称为回溯。

广度优先搜索 (breadth-first search) 先访问当前深度的所有结点，然后才移向下一个深

度。在图5-2中，它已经探索了树的前三层。由于分支数目有限，因此能够到达所有结点：这个策略是完全的。然而，除了一些很简单的例子，它却不怎么实用。因为要想到达某个给定的深度，它需要访问指数个数的结点，并耗费指数数量的空间。

5.17 用ML实现的搜索策略

无穷树是可以像无穷表那样表示的，也就是用ML的数据类型，其中包括一个延迟求值的函数。但是，对于本节中的搜索树，一个结点的子树可以通过它的标签来计算。类型 τ 上的树（具有有限分支数）是由函数 $next : \tau \rightarrow \tau list$ 来表示的，其中 $next\ x$ 是 x 的子树表。

通过利用堆栈来存储将要访问的结点，可以有效地实现深度优先搜索。每一步都将表头 y 移出堆栈，换成它的子树 $next\ y$ ，这棵子树就会在堆栈中的其他结点之前被访问。结点是按照被访问的顺序记录在输出序列中的。

```
fun depthFirst next x =
  let fun dfs []      = Nil
      | dfs(y::ys) = Cons(y, fn()=> dfs(next y @ ys))
  in dfs [x] end;
> val depthFirst = fn : ('a -> 'a list) -> 'a -> 'a seq
```

广度优先搜索将等待访问的结点存放在队列中，而不是堆栈中。当 y 被访问以后， $next\ y$ 中的子结点会被放在队尾。^①

```
fun breadthFirst next x =
  let fun bfs []      = Nil
      | bfs(y::ys) = Cons(y, fn()=> bfs(ys @ next y))
  in bfs [x] end;
> val breadthFirst = fn : ('a -> 'a list) -> 'a -> 'a seq
```

两个策略都是简单地以某种顺序枚举所有的结点。解是通过算子 $Seq.filter$ 将某个适当的谓词作用到结点上来的确定的。其他的搜索策略也可以通过修改这两个函数来得到。

❶ 最佳优先搜索 (best-first search)。人工智能中的搜索经常使用启发式距离函数，这种函数可以估计任意给定结点到解的距离。这个估计被加到已知的该结点到根结点的距离上，由此估计出从根经由该结点到解的距离。这些估计将一个顺序强加到等待搜索的结点上，而这些结点则存放在一个优先队列中。具有最小估计总距离的结点就是下一个要访问的。

如果距离函数相当精确，最佳优先搜索会很快地收敛到一个解上。如果这是一个常函数，那么最佳优先搜索就会退化成广度优先搜索。如果它错误地估计了真实的距离，那么最佳优先搜索可能永远也找不到任何解。这个策略有很多种形式，其中最简单的是A*算法。更多资料参见Rich和Knight (1991)。

204

206

练习 5.39 改写 $depthFirst$ 和 $breadthFirst$ ，增加一个参数：判断解的谓词。这比课文中介绍的办法稍微高效一些，因为它避免了对 $Seq.filter$ 的调用和对输出序列的复制。

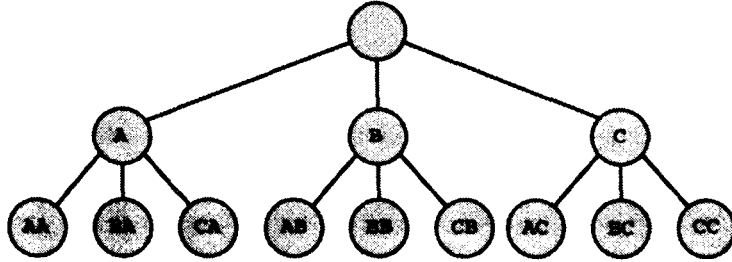
练习 5.40 像上文介绍的那样实现最佳优先搜索。你的函数必须跟踪每一个结点到根结点的

① 堆栈和队列在这里被表示为表。表可以实现高效的堆栈，但用表实现队列就很糟了。7.3节展示了高效的队列。

距离，以便将它加到从该结点到解的估计距离上。

5.18 生成回文

让我们生成字母集{A, B, C}上的回文序列。搜索树中的每一个结点将是这三个字母组成的表*l*，并有3个分支分别指向结点#“A”:::1、#“B”:::1和#“C”:::1。



函数*nextChar*生成了这样一棵树。

```
fun nextChar l = [#"A":::l, #"B":::l, #"C":::l];
> val nextChar = fn : char list -> char list list
```

回文 (palindrome) 是和自己的翻转表相等的表。让我们声明相应的谓词:

```
fun isPalin l = (l = rev l);
> val isPalin = fn : 'a list -> bool
```

当然，有很多更高效的办法来生成回文。我们的方法旨在强调不同搜索策略间的不同点。首先声明一个函数来帮助显示结点序列 (*implode*将字符表连成一个字符串):

```
fun show n csq = map implode (Seq.take(csq,n));
> val show = fn : int -> char list seq -> string list
```

广度优先搜索是完全的，可以生成所有的回文。让我们在过滤前后检查一下这个序列:

207

```
show 8 (breadthFirst nextChar []);
> ["", "A", "B", "C", "AA", "BA", "CA", "AB"] : string list
show 8 (Seq.filter isPalin (breadthFirst nextChar []));
> ["", "A", "B", "C", "AA", "BB", "CC", "AAA"] : string list
```

深度优先搜索则不能找到所有解。因为树的最左分支是无穷的，所以搜索永远也离不开这个分支。我们甚至都不需要调用*Seq.filter*:

```
show 8 (depthFirst nextChar []);
> ["", "A", "AA", "AAA", "AAAA", "AAAAA", "AAAAAA",
> "AAAAAAA"] : string list
```

如果在一个无穷分支上没有解，那么深度优先搜索根本找不到任何东西。让我们从标签*B*开始搜索。只有一条回文具有形式*AA...AB*:

```
show 5 (depthFirst nextChar [#"B"]);
> ["B", "AB", "AAB", "AAAB", "AAAAB"] : string list
```

如果尝试在这个序列中发现一条以上的回文

```
show 2 (Seq.filter isPalin (depthFirst nextChar [#"B"]));
```

……它将永远运行下去。

另一方面，广度优先搜索探索了 B 以下的整个子树。过滤序列产生了所有以 B 结尾的回文：

```
show 6 (breadthFirst nextChar ["B"]);
> ["B", "AB", "BB", "CB", "AAB", "BAB"] : string list
show 6 (Seq.filter isPalin (breadthFirst nextChar ["B"]));
> ["B", "BB", "BAB", "BBB", "BCB", "BAAB"] : string list
```

我们再次看到了完全搜索策略的重要性。

5.19 八皇后问题

有一个经典的问题，要求在一个国际象棋棋盘上放置8个皇后，使得任何一个皇后都不能攻击另一个。也就是说，任何两个皇后不能在同一行，同一列或同一斜线上。可以通过考察所有将皇后逐列安全放置的办法来求解。搜索树的根结点是一个空棋盘。在第一列一共有8个位置可以放置皇后，因此从根结点到有一个皇后的棋盘有8个分支。一旦有一个皇后被放置在第一列，第二列可以安全放置皇后的位置就会少于8个；分支数目随着树的深入而减少。含有8个皇后的棋盘必然是一个叶子。

208

由于树是有限的，因此深度优先搜索可以找到所有的解。大多数发表了的解法，不论是过程式的还是函数式的，都是直接编码深度优先搜索的。过程式程序，通过将皇后所占用的行和斜线存储在布尔数组里面，可以迅速地找到所有解。这里，八皇后问题只是作为不同搜索策略的演示。

我们可以将含有皇后的棋盘表示为行号的表。表 $[q_1, \dots, q_k]$ 代表了这样一个棋盘，其中皇后们在第 i 列的第 q_i 行， $i = 1, \dots, k$ 。函数`safeQueen`测试一个皇后是否能安全地放置在下一列的第`newq`行，然后形成新的棋盘 $[newq, q_1, \dots, q_k]$ 。（之前的列只是被左移了一列。）新放置的皇后不能和原有的其他皇后在同一行或同一斜线上。注意 $|newq - q_i| = i$ 仅当`newq`和 q_i 在同一斜线上。

```
fun safeQueen oldqs newq =
  let fun nodiag (i, []) = true
      | nodiag (i, q::qs) =
          Int.abs(newq-q) <> i andalso nodiag (i+1, qs)
  in not (newq mem oldqs) andalso nodiag (1, oldqs) end;
```

为了生成搜索树，函数`nextQueen`以一个现有的棋盘为参数，返回所有安全加入了新皇后的棋盘列表。观察一下表算子的使用，通过`map`应用了一个切片，并通过`filter`应用了一个柯里函数。八皇后问题也推广到了 n 皇后，也就是将 n 个皇后安全地放置在 $n \times n$ 的棋盘上。通过调用`upto`（在3.1节声明）可以生成 $[1, \dots, n]$ 的候选皇后表。

```
fun nextQueen n qs =
  map (secur op:: qs) (List.filter (safeQueen qs) (upto(1,n)));
> val nextQueen = fn : int -> int list -> int list list
```

现在定义一个谓词来判断解。由于只会考虑安全位置，解就是任何一个包含了 n 个皇后的棋盘。

```
fun isFull n qs = (length qs=n);
> val isFull = fn : int -> 'a list -> bool
```

函数`depthFirst`可以找到八皇后全部的92个解。这需要130毫秒:

```
fun depthQueen n = Seq.filter (isFull n) (depthFirst (nextQueen n) []);
> val depthQueen = fn : int -> int list seq
Seq.toList (depthQueen 8);
> [[4, 2, 7, 3, 6, 8, 5, 1], [5, 2, 4, 7, 3, 8, 6, 1],
> [3, 5, 2, 8, 6, 4, 7, 1], [3, 6, 4, 2, 8, 5, 7, 1],
> [5, 7, 1, 3, 8, 6, 4, 2], [4, 6, 8, 3, 1, 7, 5, 2],
> ...] : int list list
```

由于序列是惰性的,因此可以逐个地求出所需的解。深度优先搜索很快就找到了第一个解(6.6毫秒)。虽然对于八皇后问题来说这样做的意义不大,但是十五皇后问题的解超过两百万个。我们可以在一秒钟之内计算出其中的几个:

209

```
Seq.take (depthQueen 15, 3);
> [[8, 11, 7, 15, 6, 9, 13, 4, 14, 12, 10, 2, 5, 3, 1],
> [11, 13, 10, 4, 6, 8, 15, 2, 12, 14, 9, 7, 5, 3, 1],
> [13, 11, 8, 6, 2, 9, 14, 4, 15, 10, 12, 7, 5, 3, 1]]
> : int list list
```

想像一下可以按照需要生成解的过程式程序该怎样设计。这很可能涉及到协同例程(coroutines)或通信进程(communicating processes)。

函数`breadthFirst`要慢些才能找到解。[⊖]而且找到一个解所花的时间和找到所有解差不多!因为解都出现在搜索树的同一深度;找到第一个解基本上需要遍历整棵树。

5.20 迭代深化

深度优先迭代深化(depth-first iterative deepening)结合了其他搜索过程的优点。像深度优先搜索那样只需要很少的空间;同时又像广度优先那样是完全的。这个策略反复地搜索一棵树,直到某一有限但又不断增长的深度。首先这个策略进行深度优先搜索到某个深度 d ,返回所有找到的解。然后再进行搜索到深度 $2d$,返回所有在深度 d 和 $2d$ 之间找到的解。然后再搜索到深度 $3d$,等等。由于每次搜索都是有限的,这个策略最终会到达任何深度。[⋄]

重复地搜索并不像初看上去那样浪费。迭代深化所增加的到达某一深度所用的时间不会超过某一个常数因子,除非搜索树分支非常少。深度 kd 和 $(k+1)d$ 之间的结点要比 kd 之上的结点多(Korf, 1985)。

为了简单起见,让我们实现 $d = 1$ 的迭代深化。它将产生和广度优先搜索一样的结果,不同的是它需要的时间较长,但节省很多空间。

函数`depthFirst`不太容易通过修改来实现迭代深化,因为它的堆栈包括了来自树的不同深度的结点。下面的搜索函数不使用堆栈;它使用单独的递归调用来访问每一棵子树。`dfs`的参数`sf`积累了(可能是无穷的)解的序列。

```
fun depthIter next x =
  let fun dfs k (y, sf) =
        if k=0 then fn()=> Cons(y,sf)
        else foldr (dfs (k-1)) sf (next y)
      fun deepen k = dfs k (x, fn()=> deepen (k+1)) ()
    in deepen 0 end;
> val depthIter = fn : ('a -> 'a list) -> 'a -> 'a seq
```

⊖ 它需要310毫秒,使用高效队列的版本需要160毫秒。

让我们详细地考察一下这个声明。必须使用尾函数（具有类型 $unit \rightarrow \alpha \text{ seq}$ ），而不能使用序列，主要是为了延时求值。函数调用 $dfs\ k\ (y, sf)$ 构造了一个序列，里面包含了在结点 y 之下深度 k 处找到的所有解，这个序列后面还跟随着序列 $sf()$ 。这里有两种情形需要考虑。

1. 如果 $k = 0$ ，那么就把 y 放在输出结果中。

2. 如果 $k > 0$ ，那么设 $next\ y = [y_1, \dots, y_n]$ 。这些结点，也就是 y 的子树，通过 $foldr$ 来递归处理。结果序列包括了所有在 y_1, \dots, y_n 以下深度 $k - 1$ 处的解：

$$dfs\ (k - 1)\ (y_1, \dots\ dfs\ (k - 1)\ (y_n, sf)\ \dots)\ ()$$

调用 $deepen\ k$ 建立一个尾函数来计算 $deepen\ (k + 1)$ ，并将其传给 dfs ， dfs 将在之前插入所有在深度 k 处找到的解。

让我们在前面的例子上试验一下这个策略。迭代深化生成了和广度优先搜索同样的回文序列：

```
show 8 (Seq.filter isPalin (depthIter nextChar []));
> ["", "A", "B", "C", "AA", "BB", "CC", "AAA"] : string list
```

它也可以用来解决八皇后问题，但相当慢（340毫秒）。使用较大的深度间隔 d ，迭代深化可以恢复一些深度优先搜索的效率，同时保持搜索的完全性。

练习 5.41 $depthIter$ 中的一个错误是，在搜索空间有穷的情况下，它的探索会超过树的深度。在试图寻找八皇后问题的第93个解时，它会永远运行下去。试着改正这个错误，看看你的版本和 $depthIter$ 一样快吗？

练习 5.42 推广函数 $depthIter$ ，把深度间隔 d 作为它的参数。使用 $d = 5$ 来生成回文。和其他策略得到的结果有什么不同吗？

练习 5.43 声明一个有穷分支但可能是无穷深度的搜索树的数据类型，使用类似序列的表示方法。书写一个函数来构造根据参数 $next : \alpha \rightarrow \alpha \text{ list}$ 所生成的树。给出一个不能用这种方法生成的树的例子。

要点小结

- ML表达式的计算结果可以是一个函数。
- 柯里函数就像一个具有多个参数的函数。
- 高阶函数封装了通用的计算形式，减少了单独的函数声明。
- 惰性表可以包含无穷多个元素，但只有有穷数目的元素可以被计算出来。
- 无穷表可以连接消费者和生产者，使得项目仅在被消费时才产生。

第6章 函数式程序的论证

大多数程序员都知道让一个程序工作是多么地困难。在20世纪70年代，软件项目的复杂性增长到从未有过的程度，程序员们已经明显地应付不了了。很多系统被推迟和取消；开发成本逐步升高。为了应对那时的软件危机，出现了几种新的方法论，每一种都以掌控大型系统的复杂性为出发点。

结构化程序设计探索如何将程序组织成具有简单接口的简单部分。抽象数据类型则使程序员可以将数据结构及其操作看成是一个数学对象。下一章是关于模块的，那里会对这个主题进行更多的论述。

函数式程序设计和逻辑程序设计的目标是将计算直接用数学的方式表达。复杂的机器状态是不可见的；程序员一次只需要理解一个表达式。

本章将介绍程序正确性证明。像其他应对软件危机的方法一样，形式方法的目的也在于增强我们的理解能力。首先要知道的就是，只有当这个程序的描述是正确的时候，它才会“工作”。我们的思维并不能分析数以亿万计的运行步骤，然而，如果程序是以数学方式表达的，那么每一阶段的计算就都可以用一个公式来描述。程序可以被验证——证明是正确的——或者从它的描述中导出。大多数的早期程序验证工作都集中在Pascal和类似的语言上，而函数式语言则更为容易论证，因为它们不涉及机器状态。

本章提要

这一章讲述了函数式程序的证明，特别着重于归纳法。证明的方法是严格的，但不是正式的。证明的目的在于增加我们对程序的理解。

本章包括以下几节：

- 一些数学证明的原理。有一类ML程序可以在初等数学范围内进行处理。有些整数函数可以用数学归纳法来验证。
- 结构归纳法。这个原则把数学归纳法推广到了有限的表和树。这部分还展示了高阶函数的证明。
- 一般性归纳原理。这部分讨论了一些不寻常的归纳证明。良基归纳为此类证明提供了统一的框架。
- 描述和验证。本章所介绍的方法将应用到一个大的例子上：某合并排序函数的验证。这部分也讨论了一些程序验证的局限性。

213

一些数学证明的原理

本章的证明是以典型的离散数学方式引出的。大多数证明都是通过归纳法进行的。很多论证都涉及方程式，将一项替换成相等的另一项，当然，逻辑连接词和量词也是必不可少的。

6.1 ML程序和数学

在证明中, 将ML程序看成是数学对象, 并遵守数学定律。完整的语言理论可能过于复杂了, 因此程序的形式需要限制一下, 这里只允许使用函数式的程序, 而不允许使用ML的命令式功能。类型被解释为集合, 这将限制数据类型声明的形式。异常也是不允许的, 虽然把它放进我们的框架中并不是很困难。表达式要求是明确定义的, 必须具有合法的类型, 并且代表可以终止的计算。

如果所有计算都必须终止, 那么一定要限制递归函数的定义。回想一下函数*facti*, 它是如下定义的:

```
fun facti (n, p) =
  if n=0 then p else facti(n-1, n*p);
```

2.11节中说过这个函数式程序是通过简化来求值的:

$$\text{facti}(4, 1) \Rightarrow \text{facti}(4-1, 4 \times 1) \Rightarrow \text{facti}(3, 4) \Rightarrow \dots \Rightarrow 24$$

对于所有的 $n \geq 0$, 对*facti*(n, p)求值都会产生唯一的结果, 因此*facti*是满足下面定律的数学函数:

$$\begin{aligned} \text{facti}(0, p) &= p \\ \text{facti}(n, p) &= \text{facti}(n-1, n \times p) \end{aligned} \quad \text{当 } n > 0 \text{ 时}$$

如果 $n < 0$, 那么*facti*(n, p)将永远计算下去, 也就是无定义的。可以认为*facti*(n, p)仅当 $n \geq 0$ 时才是有意义的, 后者是函数的前提条件 (precondition)。

214

再举一例, 考虑下面的声明:

```
fun undef (x) = undef (x)-1;
```

由于对于任何 x 来说, *undef* (x)都是不能终止的, 应该认为它是没有意义的。同样, 也不能将*undef* (x) = *undef* (x) - 1作为一条关于数的定律, 因为它显然是错的。

我们可以引入值 \perp (称为“底”, bottom) 来表示不能终止的计算的结果, 并开发一种论域理论 (domain theory) 来论证任意的递归函数定义。论域将*undef*解释成为一个对所有 x 都满足*undef* (x) = \perp 的函数。论域理论规定了 $\perp - 1 = \perp$, 所以*undef* (x) = *undef* (x) - 1的意思只是 $\perp = \perp$, 这是有效的。不过, 论域理论是复杂且难懂的。值 \perp 引入了对于所有类型的偏序, 理论中的所有函数在这个偏序上都必须单调和连续的, 递归函数则表示了最小不动点。通过强制函数必须是可终止的, 我们可以将讨论限制在初等集合论的范围内。

将我们限制在可终止计算上必须承受一些牺牲, 想要论证不总是终止的程序就更难了, 比如像解释器这种程序。遗憾的是我们也无法论证惰性求值, 因为使用这种复杂的函数式程序设计形式需要数学上的洞察力。大多数函数式程序员最终都会学习一些论域理论, 因为没有其他办法去理解无穷表上的计算到底意味着什么。

逻辑符号。本章假设你对形式证明有一定的了解。我们将采用下面的符号来表达逻辑公式:

$\neg \phi$	ϕ 的否定
$\phi \wedge \psi$	ϕ 且 ψ
$\phi \vee \psi$	ϕ 或 ψ
$\phi \rightarrow \psi$	ϕ 蕴涵 ψ
$\phi \leftrightarrow \psi$	ϕ 当且仅当 ψ

$\forall x. \phi(x)$ 对于任意 x 有 $\phi(x)$

$\exists x. \phi(x)$ 存在 x 有 $\phi(x)$

从最高优先级到最低优先级的连接词排列是 \neg 、 \wedge 、 \vee 、 \rightarrow 、 \leftrightarrow 。下面是公式里面优先级的例子：

$P \wedge Q \rightarrow P \vee \neg R$ 是 $(P \wedge Q) \rightarrow (P \vee (\neg R))$ 的缩写

量词的作用域是尽量向右扩展的：

$\forall x. P \wedge \exists y. Q \rightarrow R$ 是 $\forall x. (P \wedge (\exists y. (Q \rightarrow R)))$ 的缩写

215

很多逻辑学家喜欢另一种稍有不同省略点号的量词记法。那样的话就变成 $\forall x P \wedge \exists y Q \rightarrow R$ 是 $((\forall x P) \wedge (\exists y Q)) \rightarrow R$ 的简写。我们的这种非传统写法也很适合用在证明里，因为典型公式的量词都是出现在前面的。

连接词和量词是用来构造公式的，它们并不能取代汉语。我们也可以写“对于所有 x ，公式 $\forall y \phi(x, y)$ 为真”。

❶ 基础阅读。很多离散数学课本都讲到了谓词逻辑和归纳法。Mattson (1993) 中对这两个主题都有广泛的论述。Reeves和Clarke (1990) 则很少提及归纳法，但详细讲述了逻辑，还包括一章是关于自然演绎法的。Winskel (1993) 中包含了关于基本逻辑，归纳法和论域理论的一些章节。Gunter (1992) 讲述了论域理论以及其他关于惰性求值、ML和多态性的深入课题。

6.2 数学归纳法和完全归纳法

让我们从复习数学归纳法开始。假设 $\phi(n)$ 是我们想要证明的对所有自然数 n （这里指非负整数）都成立的性质。要想利用归纳法证明它，只要证明两件事：基本情形（base case），也就是 $\phi(0)$ ，以及归纳步骤（induction step），也就是对于所有 k ，都有 $\phi(k)$ 蕴涵 $\phi(k+1)$ 。

这个规则可以写成下面这种形式：

$$\frac{[\phi(k)] \quad \phi(0) \quad \phi(k+1)}{\phi(n)} \quad \text{限制条件：} k \text{ 不能出现在 } \phi(k+1) \text{ 的其他假设中。}$$

在这种记法中，前提写在线的上方，结论写在下方。前提包括基本情形和归纳步骤。在方括号中的公式 $\phi(k)$ 叫做归纳假设（induction hypothesis），在证明 $\phi(k+1)$ 时可以假设它成立。限制条件的意思是 k 必须是一个新的变量，没有在其他归纳假设（或其他条件）中出现过，这确保 k 可以代表任意的值。这个规则的记法来自自然演绎法（natural deduction），是一种关于证明的形式理论。

在归纳步骤中，我们在假设 $\phi(k)$ 成立的条件下证明了 $\phi(k+1)$ 。所假设的正好是要证明的性质，不过只是关于 k 的。这可能看上去像是在循环论证，特别是因为 n 和 k 通常是同一个变量（以避免一定要显式地写出归纳假设）。那为什么归纳法是合理的呢？如果基本情形和归纳步骤都成立，那么根据基本情形就有 $\phi(0)$ ，并且通过不断使用归纳步骤，相继得出 $\phi(1)$ 、 $\phi(2)$ 、…。因此 $\phi(n)$ 对于所有 n 都成立。

作为一个简单的例子，我们来证明下面的定理。

216

定理 1 每个自然数要么是偶数，要么是奇数。

证明 要归纳的性质是

n 是偶数或 n 是奇数

我们通过对 n 进行归纳来证明。

基本情形，0是偶数或0是奇数，这很显然：0是偶数。

在归纳步骤里，我们作归纳假设

k 是偶数或 k 是奇数

然后证明

$k+1$ 是偶数或 $k+1$ 是奇数

根据归纳假设，存在两种情形：如果 k 是偶数，那么 $k+1$ 是奇数；如果 k 是奇数，那么 $k+1$ 是偶数。由于两种情形下结论都成立，因此证明结束。 \square

注意，方块符号标志了证明的结尾。

这个证明不仅仅是告诉我们每个自然数不是偶数就是奇数，另外还包含了一个测试奇偶数的方法。这个测试方法可以在ML中写成一个递归函数：

```
datatype evenodd = Even | Odd;

fun test 0 = Even
  | test n = (case test (n-1) of
               Even => Odd
               | Odd  => Even);
```

在有些构造性数学的理论中，从每个归纳的证明中都可以自动提取出一个递归函数。我们不准在这研究这种理论，不过将试着从证明中尽可能多地学些东西。如果每个证明除了给我们一个公式以外没有别的东西，那么数学确实是非常乏味的。通过细化定理里面的语句，可以从证明中得到更多的信息。

定理 2 任何自然数都可以写成 $2m$ 或 $2m+1$ 的形式，其中 m 为某个自然数。

证明 由于这个性质有点复杂，我们用逻辑符号来表达它：

217

$$\exists m. n = 2m \vee n = 2m + 1$$

我们通过对 n 进行归纳来证明。

基本情形是

$$\exists m. 0 = 2m \vee 0 = 2m + 1$$

在 $m=0$ 时这个情形成立，因为 $0=2 \times 0$ 。

对于归纳步骤来说，先假定归纳假设

$$\exists m. k = 2m \vee k = 2m + 1$$

然后说明（将 m 改名为 m' 以避免混淆）

$$\exists m'. k+1 = 2m' \vee k+1 = 2m'+1$$

根据归纳假设，存在某个 m 使得 $k=2m$ 或者 $k=2m+1$ 。对于这两种情况，我们都可以举出某个

m' 使得 $k+1=2m'$ 或者 $k+1=2m'+1$ 。

- 如果 $k=2m$ ，那么 $k+1=2m+1$ ，所以 $m'=m$ 。
- 如果 $k=2m+1$ ，那么 $k+1=2m+2=2(m+1)$ ，所以 $m'=m+1$ 。

这就完成了证明。 \square

包含在这个更为详细的证明中的函数，就不仅仅是测试数的奇偶性了，它同时产生了除以二所得的商。这提供了足够的信息来重组原数，我们可以就此来检验函数结果。

```
fun half 0 = (Even, 0)
  | half n = (case half (n-1) of
               (Even, m) => (Odd, m)
             | (Odd, m)  => (Even, m+1));
```

完全归纳法。当 $k > 0$ 时，数学归纳法将问题 $\phi(k)$ 简化成子问题 $\phi(k-1)$ 。完全归纳法则将 $\phi(k)$ 简化为 k 个子问题 $\phi(0), \phi(1), \dots, \phi(k-1)$ 。它包括了数学归纳法这个特殊情形。

要对于所有 $n > 0$ 的整数证明 $\phi(n)$ 成立，则证明下面的归纳步骤就够了：

假设 $\forall i < k. \phi(i)$ 的条件下有 $\phi(k)$

218

这个归纳步骤包含了这样一个无穷的语句序列：

$\phi(0)$
 假设 $\phi(0)$ 的条件下有 $\phi(1)$
 假设 $\phi(0)$ 和 $\phi(1)$ 的条件下有 $\phi(2)$
 假设 $\phi(0)$ 、 $\phi(1)$ 和 $\phi(2)$ 的条件下有 $\phi(3)$
 \vdots

很清楚，对于所有 n ，它都蕴涵了 $\phi(n)$ ，完全归纳法因此是合理的。这个规则可以写成如下的形式：

$$\frac{[\forall i < k. \phi(i)] \quad \phi(k)}{\phi(n)} \quad \text{限制条件: } k \text{ 不能出现在前提条件的其他假设中。}$$

下面看一个简单的证明。

定理3 所有 $n > 2$ 的自然数都可以写成素数的乘积， $n = p_1 \cdots p_k$ 。

证明 对 n 进行完全归纳。这里有两种情形。

如果 n 是素数，那么结果是显然的，并且 $k=1$ 。

如果 n 不是素数，那么它可以被某个满足 $1 < m < n$ 的自然数 m 整除。由于 $m < n$ 并且 $n/m < n$ ，我们可以使用两次完全归纳法的归纳假设，把这两个数写成素数的乘积

$$m = p_1 \cdots p_k \text{ 且 } n/m = q_1 \cdots q_l$$

现在 $n = m \times (n/m) = p_1 \cdots p_k q_1 \cdots q_l$ 。 \square

这就是算术基本定理的简单部分。较难的部分是证明质因数分解的唯一性，与证明中 m 的选择无关 (Davenport, 1952)。如证明所示，它提供了一个不确定的算法来将数分解为质因数。

i 将证明作为程序。在构造性证明和函数式程序之间存在着精确的对应关系。如果我们可以从证明中提取程序，那么通过证明定理就可以得到验证好了的程序。当然，并不是每个证明都是适合的。证明不仅必须是构造性的（至少关键部分是），而且一定能描述有效率的构造。通常，自然数概念对应于一元记法（后继），这将导致不可救药的低效程序。另外，被提取出的程序还包含了逻辑论述的计算，虽然这些计算不影响结果，但是应该将它们去掉。很多人正在研究这种类型的问题。

219

Thompson (1991) 和 Turner (1991) 介绍了这个研究领域。

练习 6.1 通过归纳法证明基本的整数除法定理：如果 n 和 d 是自然数且 $d \neq 0$ ，那么存在自然数 q 和 r 使得 $n = dq + r$ 且 $0 \leq r < d$ 。用 ML 表达对应的除法函数。它的效率怎样？

练习 6.2 说明如果 $\phi(n)$ 可以通过对 n 进行归纳来用数学归纳法证明，那么它也可以用完全归纳法证明。

练习 6.3 说明如果 $\phi(n)$ 可以通过对 n 进行归纳来用完全归纳法证明，那么它也可以用数学归纳法证明。（提示：使用另一个归纳公式。）

6.3 程序验证的简单例子

描述 (specification) 是对于程序执行所需要具有的性质的精确叙述。它指定了计算的结果，而不是方法。排序的描述规定了输出必须包含和输入相同的元素，并按照升序排列。任何的排序算法都满足这个描述。描述（至少对于目前的用途来说）并没有提到性能。

程序验证 (program verification) 的意思是证明程序满足它的描述。现实中的描述是很复杂的，这使得验证变得十分困难。下面要验证的每一个程序，其描述都十分简单：程序的结果只是关于输入的简单函数。我们将验证计算阶乘、斐波那契数和幂的 ML 函数。

这些证明的关键是为函数设计一个合适的归纳。要想有效的话，归纳假设必须可以应用到函数的某个递归调用上。我们通过函数定义、其他数学定律和归纳假设来简化基本情形和归纳步骤。幸运地话，简化后的公式将显然成立，即使不然，它也至少要提出一个引理，以便我们来事先证明。

阶乘。迭代函数 *facti* 是想要计算阶乘。让我们来证明对于所有的 $n \geq 0$, $\text{facti}(n, 1) = n!$ 。回忆一下阶乘的定义： $0! = 1$ ，并且当 $n > 0$ 时， $n! = (n - 1)! \times n$ 。在 6.1 节里重复了 *facti* 的定义。

对 $\text{facti}(n, 1) = n!$ 进行归纳的想法是行不通的，因为它没有提到 *facti* 的参数 p 。这样的归纳假设没什么用处。我们必须找到把 $\text{facti}(n, p)$ 和 $n!$ 都包括进去的一个关系，并且这个关系要蕴涵 $\text{facti}(n, 1) = n!$ ，以及可以被归纳证明。 $\text{facti}(n, p) = n! \times p$ 是个很好的尝试，不过还不够，这里指的是某个特定的 n 和 p ，但是 p 随递归调用的进行而变化着。正确的公式设计包含有一个全称量词：

220

$$\forall p. \text{facti}(n, p) = n! \times p$$

这个公式作为关于某个固定 n 的归纳假设，对于所有的 p 断言了上面的等式成立。

定理 4 对于每个自然数 n ，都有 $\text{facti}(n, 1) = n!$ 。

证明 命题将通过在下面的公式中令 $p = 1$ 直接导出，而公式

$$\forall p. \text{facti}(n, p) = n! \times p$$

则通过对 n 进行归纳来证明。我们直接使用 n 取代 k ，将公式本身作为归纳假设。

对于基本情形，必须证明

$$\forall p. \text{facti}(0, p) = 0! \times p$$

这是成立的，因为 $\text{facti}(0, p) = p = 1 \times p = 0! \times p$ 。

对于归纳步骤，由于归纳假设已经在上面声明，接下来必须证明

$$\forall p. \text{facti}(n+1, p) = (n+1)! \times p$$

让我们甩掉全称量词并证明等号对于任意的 p 都成立。这里，通过将左边还原到右边来简化等式：

$$\begin{aligned} \text{facti}(n+1, p) &= \text{facti}(n, (n+1) \times p) && [\text{facti}] \\ &= n! \times ((n+1) \times p) && [\text{归纳假设}] \\ &= (n! \times (n+1)) \times p && [\text{结合律}] \\ &= (n+1)! \times p && [\text{阶乘}] \end{aligned}$$

方括号中的注释应该如下理解：

[facti]	意思是	“根据facti的定义”
[归纳假设]	意思是	“根据归纳假设”
[结合律]	意思是	“根据乘法的结合律”
[阶乘]	意思是	“根据阶乘的定义”

在归纳步骤中，等式两边被证明是相等的。注意观察归纳假设中被量词约束的变量 p 被替换成了 $(n+1) \times p$ 。□

形式证明能帮助我们理解程序。这个证明解释了 $\text{facti}(n, p)$ 中 p 的角色。要归纳的公式可以类比为过程程序验证中的循环不变式 (loop invariant)。由于证明使用了乘法的结合律，这提示我们 $\text{facti}(n, 1)$ 通过将同样一套数以不同的顺序相乘而计算出 $n!$ 。稍后，我们会将这一办法推广到一个将递归函数转换成迭代函数的定理 (6.9节)。这里，很多定理都是关于如何有效率地实现某些函数的。

斐波那契数。还记得斐波那契序列吗？它是这样定义的： $F_0 = 0$ ， $F_1 = 1$ 以及对于 $n \geq 2$ ， $F_n = F_{n-2} + F_{n-1}$ 。我们将要证明它们可以通过函数 itfib (2.15节) 计算出来：

```
fun ifib (n, prev, curr) : int =
  if n=1 then curr
  else ifib (n-1, curr, prev+curr);
```

可以看到， $\text{itfib}(n, \text{prev}, \text{curr})$ 是对所有 $n \geq 1$ 定义的，我们要证明的是 $\text{itfib}(n, 0, 1) = F_n$ 。如同上例子一样，必须推广要归纳的公式以提及函数的所有参数。并不存在一个自动的过程来实现这个目的，不过通过观察一些 $\text{itfib}(n, 0, 1)$ 的计算步骤就可以揭示出 prev 和 curr 总是斐波那契数。这个观察使我们想到了以下关系

$$\text{itfib}(n, F_k, F_{k+1}) = F_{k+n}$$

同样，在归纳之前必须加上一个全称量词。

定理 5 对于所有整数 $n \geq 1$, $itfib(n, 0, 1) = F_n$ 。

证明 我们通过对 n 进行归纳来证明下面的公式, 然后令其中 $k = 0$ 就是要证明的结果:

$$\forall k. itfib(n, F_k, F_{k+1}) = F_{k+n}$$

由于 $n \geq 1$, 因此基本情形是证明公式在 $n = 1$ 时成立:

$$\forall k. itfib(1, F_k, F_{k+1}) = F_{k+1}$$

从 $itfib$ 的定义可以立即得到这个结论。

对于归纳步骤来说, 归纳假设已经在上面给出, 接下来必须证明

$$\forall k. itfib(n + 1, F_k, F_{k+1}) = F_{k+(n+1)}$$

我们通过简化左式来证明:

$$\begin{aligned} itfib(n + 1, F_k, F_{k+1}) &= itfib(n, F_{k+1}, F_k + F_{k+1}) && [itfib] \\ &= itfib(n, F_{k+1}, F_{k+2}) && [\text{斐波那契}] \\ &= F_{(k+1)+n} && [\text{归纳假设}] \\ &= F_{k+(n+1)} && [\text{算术}] \end{aligned}$$

222

应用归纳假设的时候将 k 换成 $k + 1$, 这使用到了全称量词。 \square

这个证明显示了斐波那契数是怎样依次生成的。被归纳的公式是 $itfib$ 的一个关键性质, 但很不明显。在声明函数的时候把这样的公式作为注释标明是个很好的习惯。

幂。现在来证明对于所有实数 x 和整数 $k \geq 1$, $power(x, k) = x^k$ 。回忆一下 $power$ (2.14节) 的定义:

```
fun power(x, k) : real =
  if k=1 then x
  else if k mod 2 = 0 then power(x*x, k div 2)
        else x * power(x*x, k div 2);
> val power = fn : real * int -> real
```

这个证明将假设 ML 的实数运算是精确的, 忽略舍入误差。通常程序验证都会忽略物理硬件的局限。想要展示 $power$ 适合实际的计算机还需要一个误差分析, 所涉及的工作量要多很多。

我们必须检查 $power(x, k)$ 对于 $k \geq 1$ 是有定义的。对于 $k = 1$ 的情形, 这是显然的。如果 $k \geq 2$, 那么就需要检查函数里面的递归调用, 它们将 k 换成了 $k \text{ div } 2$ 。这一过程是会停止的, 因为 $1 < k \text{ div } 2 < k$ 。

由于 x 在 $power(x, k)$ 的计算过程中不断变化, 所以被归纳的公式必须含有一个量词:

$$\forall x. power(x, k) = x^k$$

但是普通的数学归纳法在这里就不合适了。在 $power(x, k)$ 中的递归调用将 k 换成了 $k \text{ div } 2$ 而不是 $k - 1$ 。我们要使用完全归纳法来得到关于 $k \text{ div } 2$ 的归纳假设。

定理 6 对于所有整数 $k \geq 1$, 有 $\forall x. power(x, k) = x^k$ 。

证明 通过对 k 进行完全归纳来证明这个公式。

虽然完全归纳法没有单独的基本情形，不过要对 k 的情形进行分析。由于 $k \geq 1$ ，让我们分别考虑 $k = 1$ 和 $k \geq 2$ 的情形。

当 $k = 1$ 时，要证明的是

$$\forall x. \text{power}(x, 1) = x^1$$

这是成立的，因为 $\text{power}(x, 1) = x = x^1$ 。

当 $k \geq 2$ 时，再来考虑它的子情形。如果 k 是偶数则 $k = 2j$ ，如果 k 是奇数则 $k = 2j + 1$ ，这样的 j 是存在的（也即 $k \text{ div } 2$ ）。在两种子情形中都有 $1 \leq j < k$ ，因此对于 j 有归纳假设

$$\forall x. \text{power}(x, j) = x^j$$

如果 $k = 2j$ ，那么 $k \bmod 2 = 0$ ，以及

$$\begin{aligned} \text{power}(x, 2j) &= \text{power}(x^2, j) && [\text{power}] \\ &= (x^2)^j && [\text{归纳假设}] \\ &= x^{2j} && [\text{算术}] \end{aligned}$$

如果 $k = 2j + 1$ ，那么 $k \bmod 2 = 1$ ，以及

$$\begin{aligned} \text{power}(x, 2j + 1) &= x \times \text{power}(x^2, j) && [\text{power}] \\ &= x \times (x^2)^j && [\text{归纳假设}] \\ &= x^{2j+1} && [\text{算术}] \end{aligned}$$

在两种子情形中，都是通过以 x^2 替换 x 来应用归纳假设的。 □

练习 6.4 验证`introot`计算了整数平方根（2.16节）。

练习 6.5 回忆一下2.17节的`sqroot`，它是通过牛顿-拉夫森方法来计算实数平方根的。讨论验证这个函数所涉及到的问题。

结构归纳法

数学归纳法依照自然数构成的方式来对所有自然数 n 证明 $\phi(n)$ 成立。虽然有无穷多个自然数，但是它们只用了两种构造方式：

- 0是一个自然数；
- 如果 k 是自然数，那么 $k + 1$ 也是。

严格地说，我们应该引入后继函数`suc`，然后将上式重写为：

- 如果 k 是自然数，那么`suc(k)`也是。

接下来，加法和其他算术函数也可以基于0和`suc`递归地定义，这两个符号其实就是ML数据类型的构造子。结构归纳法（structural induction）是数学归纳法在其他数据类型上的推广，例如表和树这样的类型。

6.4 关于表的结构归纳法

假设 $\phi(xs)$ 是我们希望证明的对于所有的表 xs 都成立的性质。设 xs 是某个类型 τ 的列表，类型为 $\tau \text{ list}$ ，要想通过结构归纳法证明 $\phi(xs)$ ，只要证明两个条件就行了：

- 基本情形是 $\phi([])$ 。
 - 归纳步骤是对于具有类型 τ 的所有元素 y 以及具有类型 τ list的表 ys 都有 $\phi(ys)$ 蕴涵 $\phi(y :: ys)$ 。
- 这里的归纳假设是 $\phi(ys)$ 。

这个规则可以写成如下形式:

$$\frac{[\phi(ys)] \quad \phi([]) \quad \phi(y :: ys)}{\phi(xs)} \quad \text{限制条件: } y \text{ 和 } ys \text{ 不能出现在 } \phi(y :: ys) \text{ 的其他假设中。}$$

结构归纳法为什么是合理的呢? 根据基本情形我们有 $\phi([])$ 成立。而根据归纳步骤, 对于所有的 y , 又有 $\phi([y])$ 成立, 也就是说结论对于所有单元素表都成立。再次使用归纳步骤, 结论对于两个元素的表也成立。继续这一过程, 则可以证明结论 $\phi(xs)$ 对于所有的 n -元素表都成立; 所有(该类型)的表最终都会被顾及到。这一规则也可以通过对表的长度应用数学归纳法来证明。

为了说明这条规则, 让我们来证明表的一个基本性质。

定理 7 所有的表都不等于自己的表尾。

证明 这条定理可以形式化为

$$\forall x. x :: xs \neq xs$$

我们通过对表 xs 应用结构归纳法来证明。

基本情形, $\forall x. [x] \neq []$, 根据表相等的定义显然成立。两个表是相等的当且仅当它们具有相同的长度且对应的元素也相等。

在归纳步骤中, 假定归纳假设

$$\forall x. x :: ys \neq ys$$

成立, 并(对于任意的 y 和 ys)证明

$$\forall x. x :: (y :: ys) \neq y :: ys$$

根据表相等的定义, 只要证明两边的表尾不等就够了: 也就是证明 $y :: ys \neq ys$ 。这就是归纳假设, 只要将其中受全称量词约束的 x 换成 y 即可。我们再次看到在归纳公式中量词是必要的。 □

这个定理并不适用于无穷表, 因为 $[1, 1, 1, \dots]$ 等于它自己的表尾。这里给出的结构归纳法规则只对有限对象成立。在论域理论中, 归纳法可以推广到无穷表, 但不是对任何公式都行! 这个限制是很复杂的, 简单地说, 结论对于无穷表成立的必要条件是: 归纳公式必须是若干等式的合取式。因此对于无穷表 $x :: xs \neq xs$ 是不能被证明的。

下面来证明第3章中的一些表函数定理。其中的每个函数对于所有输入都是可以终止的, 因为每个递归调用都使用了更短的表。

表的长度:

```
fun nlength []      = 0
  | nlength (x::xs) = 1 + nlength xs;
```

连接两个表的中缀操作符@:

```
fun []      @ ys = ys
  | (x::xs) @ ys = x :: (xs@ys);
```

朴素的翻转函数:

```
fun nrev []      = []
  | nrev (x::xs) = (nrev xs) @ [x];
```

一种高效的翻转函数:

```
fun revAppend ([], ys) = ys
  | revAppend (x::xs, ys) = revAppend (xs, x::ys);
```

长度和追加。下面是关于两个表连接以后的长度的一个明显性质。

定理 8 对于所有的表 xs 和 ys , 有 $nlength(xs @ ys) = nlength xs + nlength ys$ 。

证明 对 xs 进行结构归纳。我们不去改变这个变量的名字, 以便上面的公式可以直接作为归纳假设。

基本情形是

$$nlength([] @ ys) = nlength[] + nlength ys$$

这是成立的, 因为

$$\begin{aligned} nlength([] @ ys) &= nlength ys && [@] \\ &= 0 + nlength ys && [算术] \\ &= nlength[] + nlength ys && [nlength] \end{aligned}$$

226

关于归纳步骤, 假定上面的归纳假设, 并证明对于所有 x 和 xs , 有

$$nlength((x :: xs) @ ys) = nlength(x :: xs) + nlength ys$$

这是成立的, 因为

$$\begin{aligned} nlength((x :: xs) @ ys) &= nlength(x :: (xs @ ys)) && [@] \\ &= 1 + nlength(xs @ ys) && [nlength] \\ &= 1 + (nlength xs + nlength ys) && [归纳假设] \\ &= (1 + nlength xs) + nlength ys && [结合律] \\ &= nlength(x :: xs) + nlength ys && [nlength] \end{aligned}$$

也可以通过直接写 $1 + nlength xs + nlength ys$, 而省去其中的括号, 来避免显式使用结合律。

□

这个证明带出了插入表元素和对它们进行计数之间的对应关系。对 xs 进行归纳之所以可行, 是因为基本情形和归纳步骤都可以利用函数的定义进行简化。对 ys 进行归纳是没有结果的: 不信可以试试。

高效的表翻转。函数 $nrev$ 是表翻转的数学定义, 而 $revAppend$ 则可以高效地对表进行翻转。两个函数等价的证明类似于定理4, *facti*的正确性证明。在两个证明中, 归纳公式中的累加参

数都要通过全称量词进行约束。

定理 9 对于所有的表 xs , 都有 $\forall ys. revAppend(xs, ys) = nrev(xs) @ ys$ 。

证明 我们对 xs 进行结构归纳, 以上面的公式作为归纳假设。基本情形是

$$\forall ys. revAppend([], ys) = nrev[] @ ys$$

这是成立的, 因为 $revAppend([], ys) = ys = [] @ ys = nrev[] @ ys$ 。

归纳步骤是, 对于任意的 x 和 xs , 证明

$$\forall ys. revAppend(x :: xs, ys) = nrev(x :: xs) @ ys$$

简化等式的右边, 得到

227

$$nrev(x :: xs) @ ys = (nrev(xs) @ [x]) @ ys \quad [nrev]$$

简化左边得到

$$\begin{aligned} revAppend(x :: xs, ys) &= revAppend(xs, x :: ys) && [revAppend] \\ &= nrev(xs) @ (x :: ys) && [归纳假设] \\ &= nrev(xs) @ ([x] @ ys) && [@] \end{aligned}$$

归纳假设的使用是通过将全称量词约束的变量 ys 替换成 $x :: xs$ 来进行的。

证明完成了吗? 还没有: 两式的括号并不一致。还需要证明

$$nrev(xs) @ ([x] @ ys) = (nrev(xs) @ [x]) @ ys$$

这个公式看上去比当初设定要证明的那个复杂。接下来该怎么办呢? 仔细观察就会发现它是一个既简单而又应该成立的结论的特例: $@$ 是满足结合率的。我们只要证明

$$l_1 @ (l_2 @ l_3) = (l_1 @ l_2) @ l_3$$

这个归纳证明只不过是例行公事, 我们留作练习。 \square

按正确顺序证明的每个定理都将会显得更加整洁, 使演示变得完美。这个例子试图说明的是, 怎样发现对一个定理的需要。在验证中, 最困难的问题就是认识到需要证明什么样的性质。对结合律的需要在这里是很明显的, 但当我们被眼花缭乱的符号所迷惑时就不一定了, 而这又是很容易发生的事。

追加和翻转。现在我们来证明一个涉及表的连接和翻转的关系。

定理 10 对于所有表 xs 和 ys , 有 $nrev(xs @ ys) = nrev ys @ nrev xs$ 。

证明 对 xs 进行结构归纳。基本情形是

$$nrev([] @ ys) = nrev ys @ nrev[]$$

可以利用引理 $l @ [] = l$ 来证明它是成立的, 这个引理留作练习。

归纳步骤是

$$nrev((x :: xs) @ ys) = nrev ys @ nrev(x :: xs)$$

这是成立的, 因为

$$\begin{aligned}
nrev((x :: xs) @ ys) &= nrev(x :: (xs @ ys)) && [@] \\
&= nrev(xs @ ys) @ [x] && [nrev] \\
&= nrev ys @ nrev xs @ [x] && [\text{归纳假设}] \\
&= nrev ys @ nrev(x :: xs) && [nrev]
\end{aligned}$$

228

在 $nrev\ ys @ nrev\ xs @ [x]$ 中，我们通过省去括号隐式地使用了 $@$ 操作的结合律。 \square

上面两个定理说明了 $nrev$ ，虽然计算起来很低效，却是一个不错的翻转描述。它令证明变得很简单。一个文字上的描述，像

$$reverse\ [x_1, x_2, \dots, x_n] = [x_n, \dots, x_2, x_1]$$

最难形式化。函数 $revAppend$ 也不是一个好的描述，它太复杂了，并且它的性能对于描述是无关的。不过 $nlength$ 却是表长度的一个好描述。

练习 6.6 通过结构归纳法证明，对于所有表 xs ，有 $xs @ [] = xs$ 。

练习 6.7 通过结构归纳法证明，对于所有表 l_1 、 l_2 和 l_3 ，有 $l_1 @ (l_2 @ l_3) = (l_1 @ l_2) @ l_3$ 。

练习 6.8 证明对于所有表 xs ，有 $nrev(nrev\ xs) = xs$ 。

练习 6.9 证明对于所有表 xs 都有 $nlength\ xs = length\ xs$ 。（函数 $length$ 是在 3.4 节定义的。）

6.5 关于树的结构归纳法

在第 4 章我们研究了如下定义的二叉树：

```
datatype 'a tree = Lf
                | Br of 'a * 'a tree * 'a tree;
```

二叉树容许结构归纳。在多数情况下，对它们的处理类似于对表的处理。假设 $\phi(t)$ 是树的一个性质，其中 t 具有类型 $\tau\ tree$ 。要通过结构归纳法证明 $\phi(t)$ ，只要证明两个条件：

- 基本情形是 $\phi(Lf)$ 。
- 归纳步骤是，要证明对于所有类型为 τ 的元素 x 以及类型为 $\tau\ tree$ 的树 t_1 和 t_2 都有， $\phi(t_1)$ 和 $\phi(t_2)$ 蕴涵 $\phi(Br(x, t_1, t_2))$ 。这里有两个归纳假设： $\phi(t_1)$ 和 $\phi(t_2)$ 。

这个规则可以写成如下形式：

$$\frac{[\phi(t_1), \phi(t_2)] \quad \phi(Lf) \quad \phi(Br(x, t_1, t_2))}{\phi(t)} \quad \text{限制条件: } x, t_1 \text{ 和 } t_2 \text{ 不能出现在 } \phi(Br(x, t_1, t_2)) \text{ 的其他假设中。}$$

229

这一结构归纳规则是合理的，因为它概括了所有构造树的方法。基本情形建立了 $\phi(Lf)$ 。应用一次归纳步骤则对于所有 x 建立了 $\phi(Br(x, Lf, Lf))$ ，这概括了所有包含一个 Br 结点的树。应用两次归纳步骤建立了 $\phi(t)$ ，其中 t 是包含两个 Br 结点的树。进一步应用归纳步骤则概括了更大的树。

我们也可以通过树的标签数目进行完全归纳来证明这个规则，因为每一棵树都是有限的，并且它的子树要比本身小。一般来说，结构归纳法对于无穷的树是不成立的。

下面将证明一些有关二叉树函数的结论，这些函数来自 4.10 节。

树的标签数目：

```
fun size Lf = 0
    | size (Br(v, t1, t2)) = 1 + size t1 + size t2;
```

树的深度:

```
fun depth Lf          = 0
  | depth (Br(v,t1,t2)) = 1 + Int.max(depth t1, depth t2);
```

树的镜像:

```
fun reflect Lf          = Lf
  | reflect (Br(v,t1,t2)) = Br(v, reflect t2, reflect t1);
```

树标签的前序表:

```
fun preorder Lf          = []
  | preorder (Br(v,t1,t2)) = [v] @ preorder t1 @ preorder t2;
```

树标签的后序表:

```
fun postorder Lf          = []
  | postorder (Br(v,t1,t2)) = postorder t1 @ postorder t2 @ [v];
```

双重镜像。我们从一个简单的例子开始: 将一棵树镜像两次则得到原来的树。

定理 11 对于所有二叉树 t , 有 $\text{reflect}(\text{reflect } t) = t$ 。

证明 对 t 进行结构归纳。基本情形是

$$\text{reflect}(\text{reflect } Lf) = Lf$$

根据 reflect 的定义这是成立的: $\text{reflect}(\text{reflect } Lf) = \text{reflect } Lf = Lf$ 。

对于归纳步骤, 我们有两个归纳假设

230

$$\text{reflect}(\text{reflect } t_1) = t_1 \text{ 且 } \text{reflect}(\text{reflect } t_2) = t_2$$

而且必须证明

$$\text{reflect}(\text{reflect}(\text{Br}(x, t_1, t_2))) = \text{Br}(x, t_1, t_2)$$

我们来进行简化

$$\begin{aligned} & \text{reflect}(\text{reflect}(\text{Br}(x, t_1, t_2))) \\ &= \text{reflect}(\text{Br}(x, \text{reflect } t_2, \text{reflect } t_1)) && [\text{reflect}] \\ &= \text{Br}(x, \text{reflect}(\text{reflect } t_1), \text{reflect}(\text{reflect } t_2)) && [\text{reflect}] \\ &= \text{Br}(x, t_1, \text{reflect}(\text{reflect } t_2)) && [\text{归纳假设}] \\ &= \text{Br}(x, t_1, t_2) && [\text{归纳假设}] \end{aligned}$$

两个归纳假设都使用到了。可以看到两次 reflect 调用互相抵消了。 \square

前序和后序。如果你对前序和后序的概念不很清楚的话, 下面的定理可能会有所帮助。关于 nrev 和 $@$ 的定理10是很关键的结论, 在前面已经证明过了。

定理 12 对于所有二叉树 t , 有 $\text{postorder}(\text{reflect } t) = \text{nrev}(\text{preorder } t)$ 。

证明 对 t 进行结构归纳。基本情形是

$$\text{postorder}(\text{reflect } Lf) = \text{nrev}(\text{preorder } Lf)$$

证明这个只不过是例行公事; 两边都等于 $[]$ 。

对于归纳步骤, 我们有归纳假设

$$\text{postorder}(\text{reflect } t_1) = \text{nrev}(\text{preorder } t_1)$$

$$\text{postorder}(\text{reflect } t_2) = \text{nrev}(\text{preorder } t_2)$$

接着必须证明

$$\text{postorder}(\text{reflect}(\text{Br}(x, t_1, t_2))) = \text{nrev}(\text{preorder}(\text{Br}(x, t_1, t_2)))$$

首先简化右式:

$$\begin{aligned} & \text{nrev}(\text{preorder}(\text{Br}(x, t_1, t_2))) \\ &= \text{nrev}([x] @ \text{preorder } t_1 @ \text{preorder } t_2) && [\text{preorder}] \\ &= \text{nrev}(\text{preorder } t_2) @ \text{nrev}(\text{preorder } t_1) @ \text{nrev}[x] && [\text{定理10}] \\ &= \text{nrev}(\text{preorder } t_2) @ \text{nrev}(\text{preorder } t_1) @ [x] && [\text{nrev}] \end{aligned}$$

中间跳过了一些步骤。分别针对每一个@操作符,应用了两次定理10,并且 $\text{nrev}[x]$ 直接简化成了 $[x]$ 。

231

现在再来简化左式:

$$\begin{aligned} & \text{postorder}(\text{reflect}(\text{Br}(x, t_1, t_2))) \\ &= \text{postorder}(\text{Br}(x, \text{reflect } t_2, \text{reflect } t_1)) && [\text{reflect}] \\ &= \text{postorder}(\text{reflect } t_2) @ \text{postorder}(\text{reflect } t_1) @ [x] && [\text{postorder}] \\ &= \text{nrev}(\text{preorder } t_2) @ \text{nrev}(\text{preorder } t_1) @ [x] && [\text{归纳假设}] \end{aligned}$$

由此可见,等式的两边是相等的。 \square

计数和深度。现在来证明一个有关二叉树标签个数和深度之间关系的定律。这条定理是个不等式,这也提示了形式方法不仅仅只涉及等式。

定理 13 对于所有二叉树 t ,有 $\text{size } t \leq 2^{\text{depth } t} - 1$ 。

证明 对 t 进行结构归纳。基本情形是

$$\text{size } L_f \leq 2^{\text{depth } L_f} - 1$$

这是成立的,因为 $\text{size } L_f = 0 = 2^0 - 1 = 2^{\text{depth } L_f} - 1$ 。

在归纳步骤中,归纳假设是

$$\text{size } t_1 \leq 2^{\text{depth } t_1} - 1 \text{ 且 } \text{size } t_2 \leq 2^{\text{depth } t_2} - 1$$

并且我们必须说明

$$\text{size}(\text{Br}(x, t_1, t_2)) \leq 2^{\text{depth}(\text{Br}(x, t_1, t_2))} - 1$$

首先简化右式:

$$\begin{aligned} 2^{\text{depth}(\text{Br}(x, t_1, t_2))} - 1 &= 2^{1+\max(\text{depth } t_1, \text{depth } t_2)} - 1 && [\text{depth}] \\ &= 2 \times 2^{\max(\text{depth } t_1, \text{depth } t_2)} - 1 && [\text{算术}] \end{aligned}$$

然后证明左式小于或等于这个结果:

$$\begin{aligned} \text{size}(\text{Br}(x, t_1, t_2)) &= 1 + \text{size } t_1 + \text{size } t_2 && [\text{size}] \\ &< 1 + (2^{\text{depth } t_1} - 1) + (2^{\text{depth } t_2} - 1) && [\text{归纳假设}] \\ &= 2^{\text{depth } t_1} + 2^{\text{depth } t_2} - 1 && [\text{算术}] \\ &< 2 \times 2^{\max(\text{depth } t_1, \text{depth } t_2)} - 1 && [\text{算术}] \end{aligned}$$

232 上面,我们将取两个整数中最大值的数学函数`max`看成和库函数`Int.max`一样。□

i 有问题的数据类型。我们的这些简单方法并不能适用所有的ML类型。不妨考虑下面的数据类型声明:

```
datatype lambda = F of lambda -> lambda;
```

本章的数学是基于集合论的。由于不存在一个集合 A 和函数 $A \rightarrow A$ 的集合同构,因此我们搞不清楚这个声明的意义。在论域理论中,这个声明是可以解释的,因为存在论域 D 同构于 $D \rightarrow D$,后者是从 D 到 D 的连续(continuous)函数的论域。即便是在论域理论中,也没有找到适合论证 D 的归纳原则。这是因为该类型定义涉及到函数箭头(\rightarrow)左侧的递归。我们将不考虑涉及到函数的数据类型。

在项的类型`term` (5.11节)里引用了表:

```
datatype term = Var of string
              | Fun of string * term list;
```

类型`term`表示了项的有限集合,它满足结构归纳的原则。然而,类型中涉及到了表使得理论和证明变复杂了(Paulson, 1995, 4.4节)。

练习 6.10 形式化并证明:任何二叉树都不等于它自己的左子树。

练习 6.11 证明对于所有二叉树 t , $\text{size}(\text{reflect } t) = \text{size } t$ 。

练习 6.12 证明对于所有二叉树 t , $\text{nlength}(\text{preorder } t) = \text{size } t$ 。

练习 6.13 证明对于所有二叉树 t , $\text{nrev}(\text{inorder}(\text{reflect } t)) = \text{inorder } t$ 。

练习 6.14 定义一个函数`leaves`来计算一棵二叉树中的 L_f 结点。然后证明对于所有二叉树 t , $\text{leaves } t = \text{size } t + 1$ 。

练习 6.15 验证4.11节的函数`preord`。换句话说,证明对于所有二叉树 t , $\text{preord}(t, []) = \text{preorder } t$ 。

6.6 函数值和算子

我们的数学方法可以直接扩展到高阶函数(算子)的证明。“函数作为值”的记法对于数学家来说是非常熟悉的。例如,在集合论中,函数被看成是集合,并且和其他集合没什么区别。

我们可以证明很多关于算子的事实,而不需要增加任何规则。我们也可以引入 λ -演算的定律来对ML的`fn`记法进行论证,不过这里就不进行了。我们的方法,毫无疑问,只适用于纯函数,而不是那些有副作用的ML函数。

233

函数的相等。外延原则(law of extensionality)是说,如果对于所有 x (具有合适的类型),有 $f(x) = g(x)$,那么函数 f 和 g 就是相等的。例如,下面的三个函数是外延相等的:

```
fun double1(n) = 2*n;
fun double2(n) = n*2;
fun double3(n) = (n-1)+(n+1);
```

外延原则之所以有效是因为一个ML函数所进行的操作就是应用到实际参数上。将 f 替换成 g ,

如果它们是外延相等的话,不会影响任何 f 的应用^①。

另一种不同的相等概念,称为内涵相等 (intensional equality), 认为两个函数只有当它们的定义一样的时候才相等。我们的三个加倍函数在内涵相等的意义下是互不相同的。这个概念类似于Lisp中的函数相等,在这种语言中函数值是一段可以取出的Lisp代码。

并没有一种通用的、可计算的方法来测试两个函数是否外延相等。因此ML对于函数值是没有相等测试的。Lisp通过比较函数的内部表示来测试它们是否相等。

我们现在来证明几个关于函数复合 (中缀操作符 \circ) 和算子 map (5.7节) 的命题。

```
fun (f o g) x = f (g x);

fun map f []      = []
  | map f (x::xs) = (f x) :: map f xs;
```

复合的结合性。我们的第一个定理是显然的。它断言了函数复合满足结合律。

定理 14 对于所有函数 f 、 g 和 h (具有合适的类型), 有

$$(f \circ g) \circ h = f \circ (g \circ h)$$

证明 根据外延原则, 只要证明

$$((f \circ g) \circ h) x = (f \circ (g \circ h)) x$$

对于所有的 x 都成立即可。这是成立的, 因为

$$\begin{aligned} ((f \circ g) \circ h) x &= (f \circ g)(hx) \\ &= f(g(hx)) \\ &= f((g \circ h) x) \\ &= (f \circ (g \circ h)) x \end{aligned}$$

每一步推导都是根据复合的定义。 □

就像提到过的那样, 这个定理只对具有合适类型的函数成立; 等式必须被正确地定型。类型的限制适用于我们所有的定理, 以后就不再提及了。

表算子 map 。算子满足很多定律。下面是一个关于 map 和函数复合的定理, 它可以用于避免产生中间表。

定理 15 对于所有函数 f 和 g , 有 $\text{map } f \circ \text{map } g = \text{map}(f \circ g)$ 。

证明 根据外延原则, 当对于所有表 xs , 有

$$(\text{map } f \circ \text{map } g) \cdot xs = (\text{map}(f \circ g)) xs$$

时, 所要证明的等式成立。利用 \circ 的定义, 上式可以简化为

$$\text{map } f (\text{map } g xs) = \text{map}(f \circ g) xs$$

由于 xs 是一个表, 我们可以使用结构归纳法。上面的式子也就是归纳假设。基本情形是

① 外延原则依赖于我们对于函数是可终止的总体假设。ML区分 \perp (无定义的函数值) 和 $\lambda x. \perp$ (在应用时不能终止的函数), 虽然这两个函数应用到任何参数上都返回 \perp 。

$$\text{map } f (\text{map } g []) = \text{map } (f \circ g) []$$

这是成立的，因为两边都等于[]：

$$\text{map } f (\text{map } g []) = \text{map } f [] = [] = \text{map } (f \circ g) []$$

对于归纳步骤，假定归纳假设成立，并证明（对于任意 x 和 xs ）

235

$$\text{map } f (\text{map } g (x :: xs)) = \text{map } (f \circ g) (x :: xs)$$

直接论证即有

$$\begin{aligned} \text{map } f (\text{map } g (x :: xs)) &= \text{map } f ((g x) :: (\text{map } g xs)) && [\text{map}] \\ &= f(g x) :: (\text{map } f (\text{map } g xs)) && [\text{map}] \\ &= f(g x) :: (\text{map } (f \circ g) xs) && [\text{归纳假设}] \\ &= (f \circ g)(x) :: (\text{map } (f \circ g) xs) && [\circ] \\ &= \text{map } (f \circ g) (x :: xs) && [\text{map}] \end{aligned}$$

尽管出现了函数值，这也只是个例行的结构归纳证明。□

表算子 foldl 。算子 foldl 对表元素应用了一个2-参数的函数。回忆一下5.10节的定义：

```
fun foldl f e []      = e
  | foldl f e (x::xs) = foldl f (f(x, e)) xs;
```

如果 \oplus 是一个满足结合律的操作符，那么 $\text{foldl}(\text{op } \oplus) (y \oplus z) xs = (\text{foldl}(\text{op } \oplus) y xs) \oplus z$ 。因为，如果 $xs = [x_1, x_2, \dots, x_n]$ ，这就等价于

$$x_n \oplus \dots (x_2 \oplus (x_1 \oplus (y \oplus z))) \dots = x_n \oplus \dots (x_2 \oplus (x_1 \oplus y)) \oplus z$$

由于 \oplus 满足结合律，我们可以去掉所有的括号，将两边都简化成 $x_n \oplus \dots \oplus x_2 \oplus x_1 \oplus y \oplus z$ 。我们可以看到使用中缀操作符 \oplus ，而不是函数 f 记法的优点。现在来看看正式的证明。

定理 16 假设 \oplus 是一个中缀操作符，并满足结合律，即对于所有 x 、 y 和 z ，有 $x \oplus (y \oplus z) = (x \oplus y) \oplus z$ 。则对于所有 y 、 z 和 xs ，有

$$\forall y. \text{foldl}(\text{op } \oplus) (y \oplus z) xs = (\text{foldl}(\text{op } \oplus) y xs) \oplus z$$

证明 对表 xs 进行结构归纳。基本情形，

$$\text{foldl}(\text{op } \oplus) (y \oplus z) [] = (\text{foldl}(\text{op } \oplus) y []) \oplus z$$

是显然的，两边都可简化成 $y \oplus z$ 。

对于归纳步骤，我们证明

236

$$\text{foldl}(\text{op } \oplus) (y \oplus z) (x :: xs) = (\text{foldl}(\text{op } \oplus) y (x :: xs)) \oplus z$$

对于任意的 y 、 x 和 xs ：

$$\begin{aligned} \text{foldl}(\text{op } \oplus) (y \oplus z) (x :: xs) &= \text{foldl}(\text{op } \oplus) (x \oplus (y \oplus z)) xs && [\text{foldl}] \end{aligned}$$

$$\begin{aligned}
&= \text{foldl} (\text{op} \oplus) ((x \oplus y) \oplus z) xs && [\text{结合律}] \\
&= (\text{foldl} (\text{op} \oplus) (x \oplus y) xs) \oplus z && [\text{归纳假设}] \\
&= (\text{foldl} (\text{op} \oplus) y (x :: xs)) \oplus z && [\text{foldl}]
\end{aligned}$$

归纳假设是通过用 $x \oplus y$ 去替换全称量词约束的变量 y 来应用的。□

练习 6.16 证明 $\text{map } f (xs @ ys) = (\text{map } f xs) @ (\text{map } f ys)$ 。

练习 6.17 证明 $(\text{map } f) \circ \text{nrev} = \text{nrev} \circ (\text{map } f)$ 。

练习 6.18 声明一个二叉树算子 maptree ，要求满足下列等式（并给出证明）：

$$\begin{aligned}
(\text{maptree } f) \circ \text{reflect} &= \text{reflect} \circ (\text{maptree } f) \\
(\text{map } f) \circ \text{preorder} &= \text{preorder} \circ (\text{maptree } f)
\end{aligned}$$

练习 6.19 证明 $\text{foldr} (\text{op} ::) ys xs = xs @ ys$ 。

练习 6.20 证明 $\text{foldl } f z (xs @ ys) = \text{foldl } f (\text{foldl } f z xs) ys$ 。

练习 6.21 假设 \odot 和 e 满足对于所有 x 、 y 和 z ，有

$$x \odot (y \odot z) = (x \odot y) \odot z \quad \text{且} \quad e \odot x = x$$

令 F 为 $\text{foldr} (\text{op} \odot)$ 的简写。证明对于所有 y 和 l ，有 $(F e l) \odot y = F y l$ 。

练习 6.22 承接上一练习的 \odot 、 e 和 F 。定义函数 G 为 $G(l, z) = F z l$ 。证明对于所有 ls ，有 $\text{foldr } G e ls = F e (\text{map } (F e) ls)$ 。

一般性归纳原理

在表的结构归纳证明中，我们假定 $\phi(xs)$ 成立，然后证明 $\phi(x :: xs)$ 。通常归纳公式涉及了一个递归的表函数，如 nrev 。归纳假设， $\phi(xs)$ ，包含了关于 $\text{nrev}(xs)$ 的叙述。由于 $\text{nrev}(x :: xs)$ 是基于 $\text{nrev}(xs)$ 定义的，因此我们可以通过论证 $\text{nrev}(x :: xs)$ 来证明 $\phi(x :: xs)$ 。

表函数 nrev 对于参数的尾部进行了递归调用。这种仿照结构归纳法的递归调用被称作结构递归 (structural recursion)。然而，递归函数可以通过其他方式来缩短表。函数 maxl ，当应用到 $m :: n :: ns$ 上时，可能会在 $m :: ns$ 上调用自己：

```

fun maxl [m] : int = m
  | maxl (m::n::ns) = if m > n then maxl(m::ns)
                      else maxl(n::ns);

```

快速排序和合并排序都是将表划分成两个更小的表并递归地将它们排序。矩阵转置 (3.9节) 和高斯消元法则对通过删除行和列而得到的更小的矩阵进行递归调用。

大多数的树函数都使用结构递归：它们的递归调用涉及了一个结点的直接子树。将命题转换成否定范式的函数 nnf 就不是结构递归的了。我们在这一节将要证明关于 nnf 的定理。

结构归纳法很适合结构递归函数。对于其他函数，良基归纳法 (well-founded induction) 通常更优胜。良基归纳是完全归纳的一个有力推广。由于它的规则是抽象的，并且很少用于完全一般的情况下，因此我们的证明将通过一个特殊情形来进行：对大小进行归纳。例如，函数 nlength 对表的大小进行了形式化。在归纳步骤中，我们需要证明 $\phi(xs)$ ，依据的归纳假设是

$$\forall ys. nlength\ ys < nlength\ xs \rightarrow \phi(ys)$$

因此倘若表 ys 比 xs 短, 我们只要假设 $\phi(ys)$ 成立就行了。

6.7 计算范式

我们的重言式检测器使用函数来计算命题的范式 (4.19节)。这些函数涉及到了不寻常的递归; 结构归纳法看来不太合适了。首先让我们来重温一些定义。

```
datatype prop = Atom of string
              | Neg of prop
              | Conj of prop * prop
              | Disj of prop * prop;
```

函数 nnf 计算一个命题的否定范式。它实际上直接叙述了这个范式的重写规则, 因而包含了复杂的模式。

238

```
fun nnf (Atom a)           = Atom a
  | nnf (Neg (Atom a))     = Neg (Atom a)
  | nnf (Neg (Neg p))      = nnf p
  | nnf (Neg (Conj(p,q))) = nnf (Disj (Neg p, Neg q))
  | nnf (Neg (Disj(p,q))) = nnf (Conj (Neg p, Neg q))
  | nnf (Conj(p,q))        = Conj (nnf p, nnf q)
  | nnf (Disj(p,q))        = Disj (nnf p, nnf q);
```

相互递归的函数 $nnfpos$ 和 $nnfneg$ 计算同样的范式, 不过更加高效:

```
fun nnfpos (Atom a) = Atom a
  | nnfpos (Neg p)   = nnfneg p
  | nnfpos (Conj(p,q)) = Conj (nnfpos p, nnfpos q)
  | nnfpos (Disj(p,q)) = Disj (nnfpos p, nnfpos q)
and nnfneg (Atom a) = Neg (Atom a)
  | nnfneg (Neg p)   = nnfpos p
  | nnfneg (Conj(p,q)) = Disj (nnfneg p, nnfneg q)
  | nnfneg (Disj(p,q)) = Conj (nnfneg p, nnfneg q);
```

我们必须证实这些函数是可以终止的。函数 $nnfpos$ 和 $nnfneg$ 是结构递归的, 也就是说, 递归总是应用在参数的直接组成部分上, 因此它们是可以终止的。对于 nnf 来说, 终止就不是那么明显的了。考虑一下 $nnf(Neg(Conj(p, q)))$, 它的递归调用是作用在更大的表达式上的。不过, 经过几步之后就会简化成

$$Disj(nnf(Neg\ p),\ nnf(Neg\ q))$$

因此, 在 $Neg(Conj(p, q))$ 之后的递归调用涉及的是较小的命题 $Neg\ p$ 和 $Neg\ q$ 。另一个较复杂的模式, $Neg(Disj(p, q))$, 也与此类似。在每一种情况下, nnf 中的递归计算所涉及到的命题都会越来越小, 因此它是可以终止的。

我们来证明 $nnfpos$ 和 nnf 是相等的。刚才对于终止的讨论提示了关于 $nnf\ p$ 的定理应该通过对 p 的大小进行归纳而证明。我们用 $nodes(p)$ 来表示 p 里面的 Neg 、 $Conj$ 和 $Disj$ 结点的个数。这个函数在ML里面不难编写。

定理 17 对于所有命题 p , 有 $nnf\ p = nnfpos\ p$ 。

证明 对 $nodes(p)$ 进行完全归纳, 归纳假设是, 对于所有 q , 只要 $nodes(q) < nodes(p)$ 就有 $nnf\ q = nnfpos\ q$ 。对应于 nnf 的定义, 我们考虑7种情形。

如果 $p = Atom\ a$, 那么 $nnf\ (Atom\ a) = Atom\ a = nnfpos\ (Atom\ a)$ 。

如果 $p = Neg\ (Atom\ a)$, 那么

$$nnf\ (Neg\ (Atom\ a)) = Neg\ (Atom\ a) = nnfpos\ (Neg\ (Atom\ a))$$

239

如果 $p = Conj\ (r, q)$, 那么

$$\begin{aligned} nnf\ (Conj\ (r, q)) &= Conj\ (nnf\ r, nnf\ q) && [nnf] \\ &= Conj\ (nnfpos\ r, nnfpos\ q) && [\text{归纳假设}] \\ &= nnfpos\ (Conj\ (r, q)) && [nnfpos] \end{aligned}$$

情形 $p = Disj\ (r, q)$ 与此类似。

如果 $p = Neg\ (Conj\ (r, q))$, 那么

$$\begin{aligned} nnf\ (Neg\ (Conj\ (r, q))) &= nnf\ (Disj\ (Neg\ r, Neg\ q)) && [nnf] \\ &= Disj\ (nnf\ (Neg\ r), nnf\ (Neg\ q)) && [nnf] \\ &= Disj\ (nnfpos\ (Neg\ r), nnfpos\ (Neg\ q)) && [\text{归纳假设}] \\ &= nnfneg\ (Conj\ (r, q)) && [nnfneg] \\ &= nnfpos\ (Neg\ (Conj\ (r, q))) && [nnfpos] \end{aligned}$$

对 $Neg\ r$ 和 $Neg\ q$ 应用归纳假设是因为, 当用 $nodes$ 测量时, 它们比 $Neg\ (Conj\ (r, q))$ 要小。

情形 $p = Neg\ (Disj\ (r, q))$ 与此类似。

如果 $p = Neg\ (Neg\ r)$, 那么

$$\begin{aligned} nnf\ (Neg\ (Neg\ r)) &= nnf\ r && [nnf] \\ &= nnfpos\ r && [\text{归纳假设}] \\ &= nnfneg\ (Neg\ r) && [nnfneg] \\ &= nnfpos\ (Neg\ (Neg\ r)) && [nnfpos] \end{aligned}$$

这里, 因为 r 包含的结点比 $Neg\ (Neg\ r)$ 更少, 所以可以应用归纳假设。□

合取范式。现在来考虑另一个问题: 合取范式的计算是否保持了命题的含义。命题的真值指派 (truth valuation) 是一个遵守下面关系的谓词:

$$\begin{aligned} Tr\ (Neg\ p) &\leftrightarrow \neg Tr\ (p) \\ Tr\ (Conj\ (p, q)) &\leftrightarrow Tr\ (p) \wedge Tr\ (q) \\ Tr\ (Disj\ (p, q)) &\leftrightarrow Tr\ (p) \vee Tr\ (q) \end{aligned}$$

这个谓词完全由其对原子的指派 $Tr\ (Atom\ a)$ 决定。为了证明范式对所有的指派都保持了真值, 我们不对任何原子的真值作假定。

计算CNF的大多数工作是由 $distrib$ 完成的:

240

```

fun distrib (p, Conj(q, r)) = Conj(distrib(p, q), distrib(p, r))
  | distrib (Conj(q, r), p) = Conj(distrib(q, p), distrib(r, p))
  | distrib (p, q)           = Disj(p, q) (* 没有合取式 *);

```

这个函数在情形分析和递归调用上不同往常。

前两个情形有重叠部分，也就是当 $distrib(p, q)$ 的两个参数都是合取式的时候。因为ML首先尝试匹配第一个情形，第二个情形不能简单地被当作是一个等式。好像没有什么办法能将情形完全分离，除非书写几乎所有的 $Atom$ 、 Neg 、 $Disj$ 和 $Conj$ 的相互组合：大概最少也得需要13种。为了避免这样，将 $distrib$ 的第二种情形作为一个条件等式；当 p 不具有形式 $Conj(p_1, p_2)$ 时，则有

$$distrib(Conj(q, r), p) = Conj(distrib(q, p), distrib(r, p))$$

$distrib$ 的计算可能会产生改变 p 和 q 两者之一的递归调用。计算是会终止的，因为每一个调用都会减少 $nodes(p) + nodes(q)$ 的值。我们将使用这个量来进行归纳。

$distrib(p, q)$ 的任务是计算出等价于 $Disj(p, q)$ 的命题，却又具有合取范式的形式。它的正确性可以叙述如下。

定理 18 对于所有命题 p 、 q 和真值指派 Tr ，有

$$Tr(distrib(p, q)) \leftrightarrow Tr(p) \vee Tr(q)$$

证明 我们通过对 $nodes(p) + nodes(q)$ 进行归纳来证明。归纳假设是，对于所有 p' 和 q' 满足 $nodes(p') + nodes(q') < nodes(p) + nodes(q)$ ，有

$$Tr(distrib(p', q')) \leftrightarrow Tr(p') \vee Tr(q')$$

证明中考虑了和 $distrib$ 定义中相同的那些情形。

如果 $q = Conj(q', r)$ ，那么

$$\begin{aligned}
 & Tr(distrib(p, Conj(q', r))) \\
 & \leftrightarrow Tr(Conj(distrib(p, q'), distrib(p, r))) && [distrib] \\
 & \leftrightarrow Tr(distrib(p, q')) \wedge Tr(distrib(p, r)) && [Tr] \\
 & \leftrightarrow (Tr(p) \vee Tr(q')) \wedge (Tr(p) \vee Tr(r)) && [\text{归纳假设}] \\
 & \leftrightarrow Tr(p) \vee (Tr(q') \wedge Tr(r)) && [\text{分配律}] \\
 & \leftrightarrow Tr(p) \vee Tr(Conj(q', r)) && [Tr]
 \end{aligned}$$

归纳假设根据下列事实被使用了两次：

$$\begin{aligned}
 & nodes(p) + nodes(q') < nodes(p) + nodes(Conj(q', r)) \\
 & nodes(p) + nodes(r) < nodes(p) + nodes(Conj(q', r))
 \end{aligned}$$

241

我们现在可以假设 q 不是一个 $Conj$ 。如果 $p = Conj(p', r)$ ，那么可以用与上面情形相同的论证得到结论。如果 p 和 q 都不是一个 $Conj$ ，那么

$$\begin{aligned}
 & Tr(distrib(p, q)) \leftrightarrow Tr(Disj(p, q)) && [distrib] \\
 & \leftrightarrow Tr(p) \vee Tr(q) && [Tr]
 \end{aligned}$$

也就是说结论对所有情形都成立。□

证明使用了 \vee 对 \wedge 的分配律，大家可能已经预计到了这点。*distrib* 中重叠的分情丝毫也没使证明变得复杂。正相反，这种分情使得函数的定义更简洁，分析更简单。

练习 6.23 对于类型 *prop* 的值给出并证明一个结构归纳规则。为了演示这个规则，对 *p* 进行结构归纳来证明下面的公式：

$$\text{nnf } p = \text{nnfpos } p \wedge \text{nnf}(\text{Neg } p) = \text{nnfneg } p$$

练习 6.24 对于命题定义谓词 *Isnnf*，使得 *Isnnf* (*p*) 当且仅当 *p* 是一个否定范式时成立。证明对于所有命题 *p*，有 *Isnnf* (*nnf p*)。

练习 6.25 令 *Tr* 为命题的任一真值指派。证明对于所有命题 *p*，有 *Tr* (*nnf p*) \leftrightarrow *Tr* (*p*)。

6.8 良基归纳和递归

我们对归纳的处理是严格的，足以应付到目前为止所进行的非形式证明，但是还不够形式化以使它可以成为自动的。很多归纳原则都可以形式地仅从数学归纳法导出。更为统一的途径是采用良基归纳原则，大多数其他的归纳原则都是它的特例。

良基关系。关系 $<$ 是良基的 (well-founded)，如果不存在无穷的降序链：

$$\dots < x_n < \dots < x_2 < x_1$$

例如，自然数上的“小于”关系是良基的。整数上的“小于”关系却不是良基的：存在降序链：

$$\dots < -n < \dots < -2 < -1$$

242

有理数上的“小于”关系也不是良基的；考虑

$$\dots < \frac{1}{n} < \dots < \frac{1}{2} < \frac{1}{1}$$

可以看到，必须说明关系的定义域，也就是在它之上定义的值的集合，而不能仅仅说 $<$ 是良基的。

另一个良基关系是自然数序偶的字典顺序 (lexicographic ordering)，它的定义如下

$$(i', j') <_{\text{lex}} (i, j) \quad \text{当且仅当} \quad i' < i \vee (i' = i \wedge j' < j)$$

为了说明 $<_{\text{lex}}$ 是良基的，先假设存在一个无穷的降序链

$$\dots <_{\text{lex}} (i_n, j_n) <_{\text{lex}} \dots <_{\text{lex}} (i_2, j_2) <_{\text{lex}} (i_1, j_1)$$

如果 $(i', j') <_{\text{lex}} (i, j)$ ，那么 $i' < i$ 。由于在自然数上的 $<$ 是良基的，降序链

$$\dots < i_n < \dots < i_2 < i_1$$

在某一 M 步后将到达某个常数 i ：也就是说对于所有 $n \geq M$ ，有 $i_n = i$ 。现在再考虑严格的降序链

$$\dots < j_{M+n} < \dots < j_{M+1} < j_M$$

这个必定会在某一 N 步后终止在某个常数 j 上：也就是说对于所有 $n \geq M + N$ ，有 $(i_n, j_n) = (i, j)$ 。在这时，序偶链就不再变化了，和我们开始假设它在 $<_{\text{lex}}$ 关系下是降序的相矛盾。

类似的论证可以说明关于三元组、四元组等的字典顺序都是良基的。自然数表上的字典

顺序不是良基的；它允许无穷的降序链：

$$\cdots < [1, 1, \dots, 1, 2] < \cdots < [1, 2] < [2]$$

另一类良基关系是由所谓的测度函数 (measure function) 给出的。如果 f 是一个映射到自然数的函数，那么存在如下定义的良好基关系 $<_f$

$$x <_f y \text{ 当且仅当 } f(x) < f(y)$$

很明显，如果真的存在无穷的降序链

$$\cdots <_f x_n <_f \cdots <_f x_2 <_f x_1$$

那么，必然也存在无穷的降序链

243

$$\cdots < f(x_n) < \cdots < f(x_2) < f(x_1)$$

在自然数中，这是不可能的。这里 f 通常“测度”了某种事物的大小。良基关系 $<_{length}$ 和 $<_{size}$ 根据大小分别对表和树进行比较。关于 *distrib* 的证明中对命题序偶 (p, q) 使用了测度 $nodes(p) + nodes(q)$ 。

在上面关于 $<_f$ 是良基关系的说明中，将 $<$ 替换成别的良基关系也同样适用。例如， f 可以返回使用 $<_{lex}$ 比较的自然数序偶。类似地，关于 $<_{lex}$ 关系的构造方法也可以应用到任何已有的良基关系上。有很多种从其他良基关系中构造新良基关系的办法。通常，我们都可以通过构造方法来说明一个关系是良基的，而用不着讨论降序链。

良基归纳法。令 $<$ 为某一类型 τ 上的良基关系， $\phi(x)$ 为对于具有类型 τ 的所有 x 待证明的性质。要想通过良基归纳法来证明它，只要对于所有 y 证明下面的归纳步骤：

如果对于所有 $y' < y$ 有 $\phi(y')$ ，则 $\phi(y)$ 成立。

这个规则可以写成如下形式：

$$\frac{[\forall y' < y. \phi(y')]}{\phi(y)} \quad \text{限制条件：} y \text{ 不能出现在前提条件的其他假设中。}$$

这条规则可以用反证法证明成立：如果 $\phi(x)$ 对于某个 x 不成立，那么我们就可以取得一个关于 $<$ 的无穷降序链。根据归纳步骤知 $\forall y' < x. \phi(y')$ 蕴涵 $\phi(x)$ 。如果 $\neg \phi(x)$ ，那么一定有某个 $y_1 < x$ 使得 $\neg \phi(y_1)$ 。对 y_1 重复这个讨论，可以找到某个 $y_2 < y_1$ 使得 $\neg \phi(y_2)$ 。然后我们又可以得到 $y_3 < y_2$ ，依此类推。^①

完全归纳法是这个规则的一个特例，其中 $<$ 是良基关系 $<$ (在自然数上的)。其他归纳原则都是良基归纳法对于适当选择的 $<$ 关系的特例。

自然数上的前驱关系， $m <_N n$ 仅当 $m + 1 = n$ ，明显是良基的。现在考虑在归纳假设 $\forall y' <_N y. \phi(y')$ 下证明 $\phi(y)$ 。存在两种情形：

- 如果 $y = 0$ ，那么 $y' <_N 0$ 永远不会成立，我们必须直接证明 $\phi(0)$ 成立。
- 如果 $y = k + 1$ ，那么 $y' <_N k + 1$ 仅在 $y' = k$ 时成立，所以我们可以证明 $\phi(k + 1)$ 时假设

① 无穷降序链在直觉上容易理解，但是其他良基的定义使得证明更简单。例如， $<$ 是良基的仅当每一个非空集合都有一个 $<$ -极小元。也同样存在适合构造性逻辑的定义。

$\phi(k)$ 成立。

因此，在关系 \prec_N 上的良基归纳法恰好就是数学归纳法。

结构归纳法也可以类似地得到。令 \prec_L 为表上的关系，使得 $xs \prec_L ys$ 仅当对于某个 x 有 $x :: xs = ys$ 。直观地说， $xs \prec_L ys$ 表示 xs 是 ys 的表尾。在良基关系（这很显然） \prec_L 上的归纳，产生了关于表的结构归纳。令 \prec_T 为树上的关系，使得 $t' \prec_T t$ 仅当对于某个 x 和 t'' 有 $Br(x, t', t'') = t$ 或 $Br(x, t'', t') = t$ 。在这个“是……的子树”关系上的良基归纳产生了关于树的结构归纳。

通过测度函数给出的良基关系产生了对于对象大小的归纳。在论证 *distrib* 的时候，对于序偶 (p, q) 大小的归纳让我们可以避免进行先对 q 然后对 p 的嵌套结构归纳。

良基归纳也可以模拟对于量词约束公式的归纳，例如我们曾通过数学归纳法证明了

$$\forall p. \text{facti}(n, p) = n! \times p$$

其实只要对于序偶 (n, p) 上的关系 \prec_{fst} 使用良基归纳法就足以证明 $\text{facti}(n, p) = n! \times p$ ，其中

$$(n', p') \prec_{fst} (n, p) \text{ 当且仅当 } n' + 1 = n$$

虽然很多归纳原则都可以完全从数学归纳法中导出，但是推导过程通常涉及到量词。良基归纳法使得在没有量词的逻辑中可以进行有意义的证明。

良基递归。令 \prec 为类型 τ 上的良基关系。如果 f 是以 x （具有类型 τ ）为形式参数的函数，并且 f 仅当 $y \prec x$ 时才进行递归调用 $f(y)$ ，那么 $f(x)$ 对于所有 x 都是可以终止的。在这种情况下， f 是通过在 \prec 上的良基递归（well-founded recursion）所定义的。

直观地说， $f(x)$ 能够终止是因为 \prec 没有无穷的降序链：也就是不可能有无穷的递归。对于良基递归的正式证明是复杂的；除了终止问题，它还必须说明 $f(x)$ 的定义是唯一的。

对于我们的大多数递归函数来说，良基关系是显然的。如果 $n > 0$ ，那么 $\text{fact}(n)$ 递归调用 $\text{fact}(n-1)$ ，所以 fact 是由前驱关系 \prec_N 上的良基递归所定义的。当 $\text{facti}(n, p)$ 递归调用 $\text{facti}(n-1, n \times p)$ 时，它改变了第二个参数，良基关系是 \prec_{fst} 。表函数 $nlength$ 、 $@$ 和 $nrev$ 都是在“是……的表尾”关系 \prec_L 上递归的。

对一个函数是可终止的证明提示了关于这个函数的一个有用的归纳形式，请回忆我们关于 nnf 和 $distrib$ 的证明。如果一个函数是由 \prec 上的良基递归所定义，那么它通常可以用 \prec 上的良基归纳来证明。

① 实践中的良基关系。良基关系是 Boyer 和 Moore (1988) 的定理证明机的核心，这个定理证明机也叫做 NQTHM。它接受由良基递归定义的函数，并使用精致的启发函数来选择正确的关系，以用于良基归纳。它的逻辑是没有量词的，不过就像我们所看到的，这并不是一个致命的限制。NQTHM 是现有的一个最重要的定理证明机。在数学和计算机科学的许多方面所需的证明都曾经用它来完成。在定理证明机 Isabelle 中正式开发了一种良基关系的理论 (Paulson, 1995, 第3节)。

6.9 递归程序模式

良基关系允许对程序模式的论证。假设 p 和 g 是函数，并且 \oplus 是中缀操作符，考虑下面的 ML 声明

```

fun f1(x) = if p(x) then e else f1(g x) ⊕ x;
fun f2(x,y) = if p(x) then y else f2(g x, x ⊕ y);

```

假设我们还已知良基关系 $<$ ，使得在 $p(x) = \text{false}$ 时，对于所有 x 有 $g(x) < x$ 。我们于是知道 $f1$ 和 $f2$ 都是可以终止的，并且可以对关于它们的定理作出证明。

定理 19 假设 \oplus 是中缀操作符并满足结合律和存在单位元 e ；也就是对于所有 x, y 和 z ，有

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

$$e \oplus x = x = x \oplus e$$

则对于所有 x ，我们有 $f2(x, e) = f1(x)$ 。

证明 只要证明下面的公式，然后将 y 替换成 e 即可：

$$\forall y. f2(x, y) = f1(x) \oplus y$$

通过 $<$ 上的良基归纳可以证明它成立。存在两种情形。

如果 $p(x) = \text{true}$ ，那么

$$f2(x, y) = y \quad [f2]$$

$$= e \oplus y \quad [\text{单位元}]$$

$$= f1(x) \oplus y \quad [f1]$$

246

如果 $p(x) = \text{false}$ ，那么

$$f2(x, y) = f2(gx, x \oplus y) \quad [f2]$$

$$= f1(gx) \oplus x \oplus y \quad [\text{归纳假设}]$$

$$= f1(x) \oplus y \quad [f1]$$

因为 $g(x) < x$ ，所以可以应用归纳假设。这里隐式地使用了 \oplus 的可结合性。 \square

由此可见，我们可以将递归函数 ($f1$) 利用累加器转换成迭代函数 ($f2$)。这个定理可以应用于计算阶乘。令

$$e = 1$$

$$\oplus = \times$$

$$g(x) = x - 1$$

$$p(x) = (x = 0)$$

$$< = <_N$$

则 $f1$ 是阶乘函数，而 $f2$ 就是 *facti*。这个定理是定理4的推广。

我们对程序模式所采取的论证方法要比论域理论来得简单，但普遍性要差些。在论域理论中可以很简单地证明任何形如

```

fun h x = if p x then x else h(h(g x));

```

的ML函数都满足对于所有 x 有 $h(h x) = h x$ ，而不论函数是否终止。我们的办法不能轻易地处理这个问题，使用什么良基关系来说明 h 中嵌套的递归调用是可以终止的呢？

练习 6.26 回顾函数 fst , 使得对于所有 x 和 y 有 $fst(x, y) = x$. 给出使用 fst 作为测度函数的一个良基关系的例子。

练习 6.27 考虑函数 ack :

```
fun ack(0, n) = n+1
  | ack(m, 0) = ack(m-1, 1)
  | ack(m, n) = ack(m-1, ack(m, n-1));
```

利用一个良基关系来说明 $ack(m, n)$ 对于所有的自然数 m 和 n 是有定义的。用良基归纳法证明 $ack(m, n) > m + n$ 。

练习 6.28 给出一个不是传递的良基关系的例子。证明如果 $<$ 是良基的, 那么它的传递闭包 $<^+$ 也是。

练习 6.29 考虑函数 $half$:

```
fun half 0 = 0
  | half n = half(n-2);
```

证明这个函数是由良基递归所定义的, 记住说明良基关系的定义域。

练习 6.30 证明在“是……的表尾”关系 $<_L$ 上的良基归纳等价于关于表的结构归纳。

描述和验证

排序是程序验证的好例子: 它简单, 却又琐碎。需要相当的工作才能描述什么是排序。我们之前的大多数正确性证明都是关于两个函数的等价性, 篇幅很少超过一页纸。而证明函数 $tmergesort$ 的正确性却需要占去这一节的绝大部分, 尽管如此还是忽略了很多细节。

首先, 考虑一个简单一点的描述任务: 最大公因子。如果 m 和 n 是自然数, 那么 k 是它们的 GCD 仅当 k 可以同时整除 m 和 n , 并且是满足这个条件的最大的数。已知这个描述, 就不难验证使用欧几里得算法计算 GCD 的 ML 函数了:

```
fun gcd(m, n) =
  if m=0 then n else gcd(n mod m, m);
```

最简单的途径是我们知道描述定义了一个数学函数:

$$GCD(m, n) = \max \{k \mid k \text{ 同时整除 } m \text{ 和 } n\}$$

$GCD(m, n)$ 的值是唯一定义的, 除去 $m = n = 0$ 情况外, 这时最大值不存在; 我们不需要知道 $GCD(m, n)$ 是否可计算。利用简单的数论就可以证明下面的事实:

$$\begin{aligned} GCD(0, n) &= n && \text{当 } n > 0 \text{ 时} \\ GCD(m, n) &= GCD(n \bmod m, m) && \text{当 } m > 0 \text{ 时} \end{aligned}$$

而通过简单的归纳可以证明对于所有不都为 0 的自然数 m 和 n , 有 $gcd(m, n) = GCD(m, n)$ 。从中也可以知道 $GCD(m, n)$ 是可计算的。

排序函数则不能如此验证。试图定义一个数学函数 $sorting$, 然后证明 $tmergesort(xs) = sorting(xs)$ 的做法并不实际。排序涉及了两个不同的正确性性质, 我们可以分别考虑:

1. 输出必须是一个有序表。

2. 输出必须是输入元素的某个重新排列。

在程序验证中,某些正确性性质经常被忽略。这是很危险的。函数可以返回空表来满足性质1,或将输入表原封不动的返回来满足性质2。单独的某个性质是没用的。

描述并不一定需要指定输出是唯一的。我们可能会指定编译器产生正确的代码,但不应该指定确切要生成的代码。这会使得描述过于复杂,而且妨碍了代码优化。我们可以指定数据库系统要正确地响应查询,但不应该指定确切的存储格式。

在下面的几节中将要证明 *tmergesort* 是正确的,也就是说它返回了输入元素的有序排列。让我们回顾几个来自第3章的函数。对它们可以终止的证明则留作练习。

表的工具函数 *take* 和 *drop*:

```
fun take ([], i) = []
  | take (x::xs, i) = if i>0 then x::take(xs, i-1) else [];
fun drop ([], _) = []
  | drop (x::xs, i) = if i>0 then drop(xs, i-1) else x::xs;
```

合并函数:

```
fun merge ([],ys) = ys : real list
  | merge (xs, []) = xs
  | merge (x::xs, y::ys) = if x<=y then x::merge(xs, y::ys)
                           else y::merge(x::xs, ys);
```

自顶向下的合并排序:

```
fun tmergesort [] = []
  | tmergesort [x] = [x]
  | tmergesort xs =
    let val k = length xs div 2
    in merge (tmergesort (take(xs,k)),
              tmergesort (drop(xs,k)))
    end;
```

6.10 有序谓词

谓词 *ordered* 表示了一个表中的元素在关系 $<$ 下是升序排列的。它的性质如下:

$$\begin{aligned} & \text{ordered}([]) \\ & \text{ordered}([x]) \\ & \text{ordered}(x :: y :: ys) \leftrightarrow x < y \wedge \text{ordered}(y :: ys) \end{aligned}$$

注意 $\text{ordered}(x :: xs)$ 蕴涵了 $\text{ordered}(xs)$ 。现在来证明合并两个有序表将产生另一个有序表。

定理 20 对于所有表 *xs* 和 *ys*, 有

$$\text{ordered}(xs) \wedge \text{ordered}(ys) \rightarrow \text{ordered}(\text{merge}(xs, ys))$$

证明 对 $\text{nlength } xs + \text{nlength } ys$ 的值进行归纳。

如果 $xs = []$ 或 $ys = []$, 那么根据 *merge* 的定义就得到了结论。现在假设对于某个 xs' 和 ys' , 有 $xs = x :: xs'$ 且 $ys = y :: ys'$ 。我们可以假设

$ordered(x :: xs')$ 且 $ordered(y :: ys')$

然后必须证明

$ordered(merge(x :: xs', y :: ys'))$

考虑 $x < y$ 的情形。(关于 $x > y$ 的情形是类似的, 留作练习。) 根据 $merge$ 的定义, 剩下要证明的是

$ordered(x :: merge(xs', y :: ys'))$

我们已经知道有 $ordered(xs')$, 可以应用归纳假设得到

$ordered(merge(xs', y :: ys'))$

最后我们必须证明 $x < u$, 其中 u 是 $merge(xs', y :: ys')$ 的表头元素。想确定表头元素则需要进一步的情形分析。

如果 $xs' = []$, 那么 $merge(xs', y :: ys') = y :: ys'$ 。则它的表头元素是 y , 而且我们已经假设了 $x < y$ 。

如果 $xs' = v :: vs$, 那么有两种子情形:

- 如果 $v < y$, 那么 $merge(xs', y :: ys') = v :: merge(vs, y :: ys')$ 。则表头元素是 v , 因为 $xs = x :: v :: vs$, 所以根据 $ordered(xs)$ 有 $x < v$ 。
- 如果 $v > y$, 那么 $merge(xs', y :: ys') = y :: merge(xs', ys')$ 。则表头元素是 y , 而且我们已经假设了 $x < y$ 。 □

这个证明的冗长很令人惊讶。也许 $merge$ 并不如看上去的那么直接。不管怎么说, 我们现在已经做好了准备去证明 $tmergesort$ 返回一个有序表。

定理 21 对于所有表 xs , 有 $ordered(tmergesort xs)$ 。

证明 对 xs 的长度进行归纳。如果 $xs = []$ 或 $xs = [x]$, 那么结论是显而易见的, 所以现在假设 $nlength xs > 2$ 。

250

令 $k = (nlength xs) \div 2$ 。则有 $1 \leq k < nlength xs$ 。很容易证明有下列不等式成立:

$$nlength(take(xs, k)) = k < nlength xs$$

$$nlength(drop(xs, k)) = nlength xs - k < nlength xs$$

根据归纳假设, 得到以下事实:

$ordered(tmergesort(take(xs, k)))$

$ordered(tmergesort(drop(xs, k)))$

由于 $merge$ 的两个参数都是有序的, 根据上一条定理就得出了我们要的结论。 □

练习 6.31 补充本节证明中的细节。

练习 6.32 书写另一个谓词来定义有序表的概念, 并证明它等价于 $ordered$ 。

6.11 通过多重集合表示重新排列

如果排序的输出是输入序列的一个重新排列, 那么存在一个函数, 称作排列 (permutation), 将元素的输入位置映射到对应的输出位置。为了证明排序的正确性, 可以提

供一个方法来展示这个排列。然而，我们不需要这么多的信息，这会增加证明的复杂性。这样的描述过于具体了。

我们可以证明排序的输入和输出包含了一样的元素集合，而不去理会每个元素被移到什么位置。不幸的是，这个方案接受[1,1,1,1,2]作为[2,1,2]的有效排序。集合是不区分重复元素的。这种描述又过于抽象了。

多重集合 (multiset) 是描述排列的好方法。多重集合是关心元素个数而不是元素顺序的集合。多重集合<1, 1, 2>和<1, 2, 1>是一样的，但它们不同于<1, 2>。多重集合通常称为包 (bag)，这么叫的原因也是很明显的。下面是一些构成多重集合的方法：

- \emptyset ，空包，不包含任何元素。
- $\langle u \rangle$ ，单元素包，只包含一个 u 。
- $b_1 \uplus b_2$ ， b_1 和 b_2 的和包，包含所有 b_1 和 b_2 中的元素（累加重复的元素）。

251

我们不把包看作是基本概念，而是把它们作为映射到自然数的函数。如果 b 是一个包，那么 $b(x)$ 就是元素 x 在 b 中出现的个数。因此，对于所有 x ，有

$$\begin{aligned}\emptyset(x) &= 0 \\ \langle u \rangle(x) &= \begin{cases} 0 & \text{当 } u \neq x \text{ 时} \\ 1 & \text{当 } u = x \text{ 时} \end{cases} \\ (b_1 \uplus b_2)(x) &= b_1(x) + b_2(x)\end{aligned}$$

下面的定律也容易验证：

$$\begin{aligned}b_1 \uplus b_2 &= b_2 \uplus b_1 \\ (b_1 \uplus b_2) \uplus b_3 &= b_1 \uplus (b_2 \uplus b_3) \\ \emptyset \uplus b &= b\end{aligned}$$

我们来定义将表转换成包的函数：

$$\begin{aligned}\text{bag}[] &= \emptyset \\ \text{bag}(x :: xs) &= \langle x \rangle \uplus \text{bag } xs\end{aligned}$$

最终得以描述“重新排列”的正确性性质：

$$\text{bag}(\text{tnergesort } xs) = \text{bag } xs$$

预备性证明。为了说明关于多重集合的论证，让我们看一个证明。这只是个常规的归纳。^①

定理 22 对于所有表 xs 和整数 k ，有

$$\text{bag}(\text{take}(xs, k)) \uplus \text{bag}(\text{drop}(xs, k)) = \text{bag } xs$$

证明 对表 xs 进行结构归纳。在基本情形中，

$$\begin{aligned}\text{bag}(\text{take}([], k)) \uplus \text{bag}(\text{drop}([], k)) \\ = \text{bag}[] \uplus \text{bag}[]\end{aligned}\quad [take, drop]$$

① 如果我们采用标准库的 $take$ 和 $drop$ 定义的话，这个证明就不那么常规了。标准库中的 $take(xs, k)$ 可能抛出异常，除非 $0 \leq k \leq \text{length } xs$ 。我们将不得不把 k 的约束写在定理的条文里面，并据此对证明作修改。

$$\begin{aligned}
&= \emptyset \uplus \emptyset && [bag] \\
&= \emptyset && [\uplus] \\
&= bag[] && [bag] \quad \boxed{252}
\end{aligned}$$

对于归纳步骤，必须证明

$$bag(take(x :: xs, k)) \uplus bag(drop(x :: xs, k)) = bag(x :: xs)$$

如果 $k > 0$ ，那么

$$\begin{aligned}
&bag(take(x :: xs, k)) \uplus bag(drop(x :: xs, k)) \\
&= bag(x :: take(xs, k - 1)) \uplus \\
&\quad bag(drop(xs, k - 1)) && [take, drop] \\
&= \langle x \rangle \uplus bag(take(xs, k - 1)) \uplus \\
&\quad bag(drop(xs, k - 1)) && [bag] \\
&= \langle x \rangle \uplus bag\ xs && [\text{归纳假设}] \\
&= bag(x :: xs) && [bag]
\end{aligned}$$

如果 $k \leq 0$ ，那么

$$\begin{aligned}
&bag(take(x :: xs, k)) \uplus bag(drop(x :: xs, k)) \\
&= bag[] \uplus bag(x :: xs) && [take, drop] \\
&= \emptyset \uplus bag(x :: xs) && [bag] \\
&= bag(x :: xs) && [\uplus]
\end{aligned}$$

因此，对于所有整数 k 结论都成立。 \square

下一步要证明 *merge* 将其参数中的元素合在一起构成了结果。

定理 23 对于所有表 xs 和 ys ，有 $bag(merge(xs, ys)) = bag\ xs \uplus bag\ ys$ 。

证明 对 $nlength\ xs + nlength\ ys$ 的值进行归纳。

如果 $xs = []$ 或 $ys = []$ ，那么立即可以得到结论，所以现在假设对于某个 xs' 和 ys' ，有 $xs = x :: xs'$ 以及 $ys = y :: ys'$ 。我们必须证明

$$bag(merge(x :: xs', y :: ys')) = bag(x :: xs') \uplus bag(y :: ys')$$

如果 $x \leq y$ ，那么

$$\begin{aligned}
&bag(merge(x :: xs', y :: ys')) \\
&= bag(x :: merge(xs', y :: ys')) && [merge] \\
&= \langle x \rangle \uplus bag(merge(xs', y :: ys')) && [bag] \\
&= \langle x \rangle \uplus bag\ xs' \uplus bag(y :: ys') && [\text{归纳假设}] \\
&= bag(x :: xs') \uplus bag(y :: ys') && [bag]
\end{aligned}$$

$x > y$ 的情形是类似的。 \square

最后，我们证明合并排序保留了已知包中的所有元素。 $\boxed{253}$

定理 24 对于所有表 xs , 有 $bag(tmergesort\ xs) = bag\ xs$ 。

证明 对 xs 的长度进行归纳。唯一困难的情形是当 $nlength\ xs \geq 2$ 时。如同在定理 21 中一样, 归纳假设可以应用到 $take(xs, k)$ 和 $drop(xs, k)$ 上:

$$\begin{aligned} bag(tmergesort(take(xs, k))) &= bag(take(xs, k)) \\ bag(tmergesort(drop(xs, k))) &= bag(drop(xs, k)) \end{aligned}$$

因此

$$\begin{aligned} bag(tmergesort\ xs) &= bag(merge(tmergesort(take(xs, k)), \\ &\quad tmergesort(drop(xs, k)))) \quad [tmergesort] \\ &= bag(tmergesort(take(xs, k))) \uplus \\ &\quad bag(tmergesort(drop(xs, k))) \quad [\text{定理23}] \\ &= bag(take(xs, k)) \uplus bag(drop(xs, k)) \quad [\text{归纳假设}] \\ &= bag\ xs \quad [\text{定理22}] \end{aligned}$$

到此完成了 $tmergesort$ 的验证。 □

练习6.33 请验证 \uplus 满足交换律和结合律。(提示: 回忆一下函数的外延相等。)

练习6.34 请证明插入排序保留了传给它的包中的元素。特别地, 证明下列等式:

$$\begin{aligned} bag(ins(x, xs)) &= \langle x \rangle \uplus bag\ xs \\ bag(insort\ xs) &= bag\ xs \end{aligned}$$

练习6.35 请修改合并排序以去掉重复的元素: 每个输入的元素应只在输出中恰好出现一次。形式化这一性质, 并指出验证它所需的定理。

6.12 验证的意义

现在可以宣布 $tmergesort$ 得到了验证, 但是这就意味着什么吗? 我们到底为 $tmergesort$ 建立了什么? 形式验证有三个基本的局限:

254

1. 计算模型可能太不严密了。通常硬件都是被假设为绝对可靠的。能够处理诸如算术溢出、舍入误差或空间不足等特定错误的模型也是可以设计出来的。然而, 计算机可能以出乎意料的方式发生错误。如果有人砍它一斧头将会怎么样?

2. 描述可能是不完备的或是错误的。设计的需求很难形式化, 特别是当它们反映实际应用的时候。满足一个错误的描述并不能满足客户。软件工程师们都懂得验证 (verification) (我们是否正确地制造了产品?) 和确认有效 (validation) (我们是否制造了正确的产品?) 之间的区别。

3. 证明也可能含有错误。自动定理证明能够减少但不能消除出现错误的可能性。所有人为的东西都可能含有缺陷, 甚至是我们的数学原理。这不仅仅是个哲学问题。很多错误被发现于定理证明机、证明规则以及公开出版的证明中。

除了这些基本的局限外, 还有一个实际的局限: 形式证明是冗长而乏味的。回顾本章的证明, 它们大多是很费力地在证明非常基本的东西。现在想像一下去验证一个编译程序。这个描述

将是巨型的，包含了程序设计语言的语法和语义，以及目标机器完整的指令集。这个编译器将是个很大的程序。对它的证明必须分成几部分，分别验证词法分析器、类型检测程序、中间代码生成器等。有可能时间只允许验证最值得注意的部分：比如说代码生成器，所以被“验证”过的编译器有可能因为错误的词法分析而失败。

我们也不要过于悲观。书写形式描述可以揭示出设计需求中的歧义和不一致的地方。由于设计错误要比编码错误的影响大得多，因此，即便不验证代码，书写描述也是很有价值的。很多公司花费巨大的代价去生成即使不是完全形式化的也是很严格的描述。

艰苦的验证工作也会带来收获。大多数程序都是不正确的，这时尝试去证明它通常可以查明错误所在。想体会这个，可以在本章验证的任何一个程序里面插入错误，然后再走一遍证明。证明将会失败，同时失败的地方会精确地指出在什么情况下修改后的程序会出错。

正确性证明是关于程序或系统如何运作的一个详细解释。如果证明很简单，我们可以逐行地分析，把它看作是程序执行的一系列快照。比如，归纳步骤跟踪了在递归调用时所发生的事情。大型证明可能由几百条定理组成，这些定理考察了所有的组件和子系统。

255

描述和验证产生了对程序及其任务更完整的认识。这使得对系统的信心更强。形式证明不能免去对系统测试的需要，特别是对于安全至关重要的系统。测试是唯一的办法来考察计算模型和形式描述是否准确地反映了真实情况。然而，虽然测试可以检测错误，但是它不能保证成功；它也不能让我们洞悉程序是怎样工作的。

i 进一步的阅读。Bevier等（1989）验证了一个由多个层次组成的微型计算机系统，包括软件和硬件。Avra Cohn（1989a）验证了Viper微处理器的一些正确性性质。以她的证明作为例子，Cohn（1989b）讨论了验证的基本局限。

Fitzgerald等（1995）报告了一项研究，其中两个小组互相独立地开发了一个信任网关。控制小组使用了传统的方法，而实验小组在这些方法外增加了书写形式描述。关于流水线微处理器AAMP5有特别大量的研究，这是一款为航空电子设备而设计的商业产品。它被分为两个层次进行描述，并且证明了它的一些微指令是正确的（Srivastava和Miller，1995）。这两个研究都显示了书写形式描述，不论是否伴随着形式证明，都是可以发现错误的。

由Susan Gerhart等（1994）进行的一项主要研究考察了12个涉及使用形式方法的案例。在一个著名的哲学专论中，Lakatos（1976）称，我们可以从部分的甚至是错误的证明中学到知识。

要点小结

- 很多函数式程序都可以在初等数学的范畴内给予一个含义。高阶函数也是可以处理的，但是惰性求值或无穷数据结构却超出了这个范畴。
- 对于函数可终止的证明具有和大多数其他函数证明一样的通用形式。
- 数学归纳法适用于涉及自然数的递归函数。
- 结构归纳法适用于涉及表和树的递归函数。
- 良基归纳和递归可以处理广泛的可终止计算问题。
- 程序证明需要精确和简单的描述。
- 证明可能会出错，但是通常传递了有价值的信息。

256

第7章 抽象类型和函子

大的程序应该组织成有层次的模块，这一点，每个人都认可。Standard ML的结构和签名符合这个要求。结构可以将相关的类型、值和函数声明包装在一起。签名可以指定一个结构所必须包含的组件。从第2章的复数到第5章的无穷序列，我们已经利用结构和签名的最简形式对这样的一系列例子进行了处理。

模块化的结构使得程序容易理解。更为有用的是，模块应该作为可以互相替换的组件：将一个模块换成它的改良版本不应该影响程序的其他部分。Standard ML的抽象类型（abstract type）和函子（functor）可以帮助实现这个目标。

模块可能会暴露它的内部细节。当模块被替换时，程序中依赖这些细节的其他部分就会失败。ML提供了几种方法，在声明抽象类型和相关操作的同时，隐藏类型的具体表示。

如果结构 B 依赖结构 A ，并且我们希望把 A 替换成另一个结构 A' ，那么可以编辑程序文本并重新编译程序。当 A 完全过时并可以放弃时，这样做是可以满足要求的。然而，如果 A 和 A' 都有用怎么办？比如说像表达不同精度浮点运算这样的结构。

ML允许将 B 声明成用结构作为参数的结构。然后就可以使用 $B(A)$ 和 $B(A')$ 了，也可能是同时使用。像 B 这样参数化的结构叫做函子。函子允许将 A 和 A' 看作是可以互换的部分。

模块的语言部分与类型和表达式的核心语言部分是截然不同的。它关心的是程序的组织，而不是计算本身。模块可以包含类型和表达式，但反过来却不行。主要的模块构造在核心语言中存在可以类比的部分：

结构 ~ 值
签名 ~ 类型
函子 ~ 函数

257

这个类比可以作为理解的起点，但是它并不能传达ML模块系统的全部能力。

本章提要

本章将更深入地对结构和签名进行讨论，并介绍抽象类型和函子。很多模块语言里的特性主要都是为支持函子而准备的。本章包括以下几节：

- 队列的三种表示方法。三个不同的结构实现了队列，说明了多种数据表示方法的思想。但是结构并不能隐藏队列的表示，它可能被程序的其他地方误用。
- 签名和抽象。队列结构的签名约束可以隐藏细节，它声明了一个队列的抽象类型。`abstype`声明可以更为灵活地声明抽象类型。三个队列的表示法都有它们各自的具体签名。
- 函子。函子可以把三种队列实现作为可互换的部分，首先用在测试框架中，然后用于广度优先搜索。另外一个例子是泛型矩阵运算，应用在数值和图论中。函子允许对于任意

有序类型泛化地表达字典和优先队列。

- 使用模块构造大型系统。这里讨论了一系列深入的问题：函子的多个参数、共享约束以及完全函子化的程序设计方式。新的声明形式，例如`open`和`include`，可以帮助管理大型程序中纵深的层次结构。
- 模块参考指南。系统简明地表述了完整的模块语言。

队列的三种表示方法

队列 (queue) 是这样一种序列，它的元素只能从末端追加，并且只能从首端取出。队列执行的是先进先出 (FIFO, first-in-first-out) 原则。队列提供以下的操作：

- `empty`: 空队列。
- `enq(q, x)`: 通过将`x`加到`q`的末端所得到的队列。
- `null(q)`: 测试`q`是否为空所返回的布尔值。
- `hd(q)`: `q`的首元素。
- `deq(q)`: 通过将`q`的首元素删除所得到的队列。
- `E`: 在队列为空时由`hd`和`deq`所抛出的异常。

258

队列的操作是函数式的：`enq`和`deq`创建新的队列，而不是修改原有的队列。我们将讨论几种表示队列的方法，并将它们定义为ML的结构，最后找出一种高效的表示方法。

名字`enq`和`deq`是单词进队列 (enqueue) 和出队列 (dequeue) 的简写，而`null`和`hd`则和现有的表操作名字发生冲突。由于我们将操作包装在结构中，因此可以不用顾忌冲突地使用短名字。

书写相应的签名是很简单的，不过让我们先把这个推迟到下一节。那时我们还将考虑怎样通过声明抽象类型来隐藏表示方法。

7.1 将队列表示为表

也许是最明显的，表示法1将队列维持在由其元素组成的表中。结构`Queue1`声明如下：

```
structure Queue1 =
  struct
    type 'a t = 'a list;
    exception E;

    val empty = [];

    fun enq(q, x) = q @ [x];

    fun null(x::q) = false
      | null _      = true;

    fun hd(x::q) = x
      | hd []    = raise E;

    fun deq(x::q) = q
      | deq []    = raise E;
  end;
```

队列的类型就是 αt ；在结构之外是 $\alpha \text{Queue1.t}$ 。 $\alpha \text{Queue1.t}$ 是类型缩写，这使得它成为 αlist 的同义词。（可以回顾2.7节中我们将`vec`作为 $\text{real} \times \text{real}$ 的同义词。）由于类型 $\alpha \text{Queue1.t}$ 的值

可以用于任何的表操作，这个类型的名称除了作为注释以外几乎没什么用处。

函数`enq`使用了追加操作，而`deq`则是利用了模式匹配。其他的队列操作都实现很简单、很高效。不过`enq(q, x)`却需要和`q`的长度成正比的时间：很不令人满意。

结构并不能隐藏信息。除了将结构声明作为一个整体并引入了复合名字外，声明一个结构和单独声明里面的项目几乎没什么区别。每个项目的作用都和单独声明时一样。结构`Queue1`没有将队列和表区分开来：

```
Queue1.deq ["We", "happy", "few"];
> ["happy", "few"] : string list
```

259

7.2 将队列表示为新的数据类型

表示法2声明了一个数据类型，它带有构造子`empty`和`enq`。现在操作`enq(q, x)`只需要常数时间，和`q`的长度无关，但是`hd(q)`和`deq(q)`却很慢。调用`deq(q)`需要将`q`的剩余元素都复制一遍。甚至连`hd(q)`也需要递归调用。

```
structure Queue2 =
  struct
    datatype 'a t = empty
        | enq of 'a t * 'a;
    exception E;

    fun null (enq _) = false
      | null empty = true;

    fun hd (enq(empty, x)) = x
      | hd (enq(q, x)) = hd q
      | hd empty = raise E;

    fun deq (enq(empty, x)) = empty
      | deq (enq(q, x)) = enq(deq q, x)
      | deq empty = raise E;
  end;
```

通过定义新的数据类型，表示法2并没什么变化。它基本上和将队列表示成一个逆向表没什么不同。那样的话：

$$\text{enq}(q, x) = x :: q$$

而`deq`则是一个移走表尾最末元素的递归函数。我们可以称之为表示法2a。

队列类型 α `Queue2.t`并不是抽象的：它是带有构造子`Queue2.empty`和`Queue2.enq`的数据类型。利用构造子进行模式匹配可以将队列的最后一个元素移出，这破坏了它的FIFO原则：

```
fun last (Queue2.enq(q, x)) = x;
> val last = fn : 'a Queue2.t -> 'a
```

这个声明误用了该数据结构。如果在程序中这样的误用遍布各处的话，那几乎不可能对队列的表示作出改变，程序将很难维护。

同样，结构没有隐藏信息。`Queue1`和`Queue2`之间的区别从外部是可见的。函数`Queue1.null`可以应用在任何的表上，而`Queue2.null`只能应用在类型为 α `Queue2.t`的值上。`Queue1.enq`和`Queue2.enq`都是函数，不过`Queue2.enq`还是一个构造子，可以出现在模式中。

260

这个数据类型声明违反了构造子名字以大写字母为首的约定(4.4节)。在小型结构的范围中这不算什么,不过输出这样的构造子就有问题了。

7.3 将队列表示为表的序偶

表示法3 (Burton, 1982) 将队列维护在一对表里面。序偶

$$([x_1, x_2, \dots, x_m], [y_1, y_2, \dots, y_n])$$

表示队列

$$x_1 x_2 \cdots x_m y_n \cdots y_2 y_1$$

这个队列包括一个前部和一个后部。后部的元素是逆序存储的,这样新元素可以很迅速地加至队尾, $enq(q, y)$ 修改队列使得:

$$(xs, [y_1, \dots, y_n]) \mapsto (xs, [y, y_1, \dots, y_n])$$

前部的元素则以正确的顺序存储,这样它们可以被迅速地从队列中移出, $deq(q)$ 修改队列使得:

$$([x_1, x_2, \dots, x_m], ys) \mapsto ([x_2, \dots, x_m], ys)$$

当前部变为空时,就将后部翻转并移至前部:

$$([], [y_1, y_2, \dots, y_n]) \mapsto ([y_n, \dots, y_2, y_1], [])$$

然后,后部又可以进一步积累元素直到前部再次变为空。如果当 $n > 1$ 时,队列不具有下面的形式

$$([], [y_1, y_2, \dots, y_n])$$

则称这个队列为标准形式(normal form)的队列。队列的操作保证了它们的结果具有标准形式。因此,检查队列的首元素并不会引起翻转。标准队列当它的首部为空时为空。

下面是这个方案的结构。队列类型被声明为带有一个构造子的数据类型,而不是类型缩写。我们对于弹性数组曾使用了类似的技术(见图4-3)。这种构造子在运行时没有开销,却能使代码中的队列和其他类型的值区分开来。

261

```
structure Queue3 =
  struct
    datatype 'a t = Queue of ('a list * 'a list);
    exception E;

    val empty = Queue([], []);

    fun norm (Queue([], tails)) = Queue(rev tails, [])
      | norm q = q;

    fun enq (Queue(heads, tails), x) = norm (Queue(heads, x::tails));

    fun null (Queue([], [])) = true
      | null _ = false;

    fun hd (Queue(x::_, _)) = x
      | hd (Queue([], _)) = raise E;
```

```

fun deq (Queue (x :: heads, tails)) = norm (Queue (heads, tails))
  | deq (Queue ([], _))             = raise E;
end;

```

函数`norm`在需要时通过翻转后部将队列变为标准形式。它会被`enq`和`deq`调用，这是因为队列在每次加入和删除元素后都必须成为标准形式。

同样，内部细节没有被隐藏起来。使用者仍可以通过构造子`Queue3.Queue`和函数`Queue3.norm`进行篡改。通过构造子`Queue`进行模式匹配可以暴露出队列是由一对表组成的。在结构内部，这种访问必不可少；但用在外部，它就可能破坏队列的FIFO原则。在外部调用`Queue3.norm`则达不到什么目的。

❶ 这个表示法的效率如何？使用翻转操作看起来代价很高。但是从队列的整个使用周期平均来看，`enq`和`deq`操作的开销是常量。每个队列元素最多有两个表构造 (::) 操作，一个是在它被追加至队列后部时，另一个是在它被移至队列前部时。

在数据结构的整个使用周期上衡量开销的做法叫做分摊 (amortized) 开销 (Cormen等, 1990)。Sleator和Tarjan (1985) 展示了另一种数据结构：自调节树，它具有很好的分摊开销。这种数据结构的主要缺点是开销的分配并不均匀。当进行标准化操作时，翻转可能引起意外的延时。

另外，分摊开销的计算是基于假定队列的使用是按命令式的方式进行的：是单线的 (single-threaded)。每次数据结构被更新后，原先的值应该丢弃。如果我们破坏这一假定，比如不断地将`deq`应用到形如

$$([x], [y_1, y_2, \dots, y_n])$$

的队列上，那么就会发生额外的标准化操作，带来更大的开销。

262

弹性数组可以表示队列，并避免上述的缺点。但是每个操作的代价会增加到和 $\log n$ 成正比，其中 n 是队列的元素数目。表示法3简单而有效，值得推荐给大多数需要函数式队列的场合。

练习 7.1 在表示法1下，通过从空队列开始应用`enq`操作，需要多少时间才能建立起一个 n -元素的队列？

练习 7.2 讨论三种表示法的相对价值。例如，是否存在某种情况使得表示法1可能比表示法3更高效。

练习 7.3 用ML编写表示法2a。

练习 7.4 表示法4使用弹性数组，用`hiext`实现`enq`，用`lorem`实现`deq`。编写表示法4，并和表示法3比较效率。

练习 7.5 传统上队列使用数组来表示，用索引指向第一个和最后一个元素。第4章的函数式数组适合这个目的吗？它和其他函数式队列相比怎样？

签名和抽象

抽象类型 (abstract type) 是具有一套操作的类型，也只有这些操作可以应用在这个类型上。类型的表示可以改变，也许能变得更为有效，但不会影响程序的其他部分。抽象类型使

得程序更容易理解和修改。队列就应该定义为抽象类型，将内部细节隐藏起来。

我们可以通过约束结构签名来限制外部对其组件的访问。还可以通过`abstype`声明来隐藏类型的表示方法。组合运用这些方法就产生了抽象结构。

7.4 队列应具有签名

虽然结构`Queue1`、`Queue2`和`Queue3`彼此不同，但是它们都实现了队列。此外，它们还共享了一致的接口，这个接口由签名`QUEUE`给出：

```
signature QUEUE =
  sig
    type 'a t                                (* 队列的类型 *)
    exception E                              (* 在hd, deq中产生的错误 *)
    val empty: 'a t                          (* 空队列 *)
    val enq  : 'a t * 'a -> 'a t             (* 加到队尾 *)
    val null : 'a t -> bool                  (* 测试空队列 *)
    val hd   : 'a t -> 'a                    (* 返回首元素 *)
    val deq  : 'a t -> 'a t                  (* 删除首元素 *)
  end;
```

签名中的每一条叫做一个描述。每个描述后面的注释是可选的，但可以使签名传递更多的信息。若想让一个结构可以成为这个签名的实例（instance），那么它至少要定义以下几项，

- 一个多态类型 αt （它并不需要允许相等测试）
- 一个异常 E
- 一个类型为 αt 的值`empty`
- 一个类型为 $\alpha t \times \alpha \rightarrow \alpha t$ 的值`enq`
- 一个类型为 $\alpha t \rightarrow \text{bool}$ 的值`null`
- 一个类型为 $\alpha t \rightarrow \alpha$ 的值`hd`
- 一个类型为 $\alpha t \rightarrow \alpha t$ 的值`deq`

我们依次看看每一个结构。在`Queue1`中，类型 αt 是 $\alpha \text{ list}$ 的缩写，并且结构中的值在这个缩写下都具有正确的类型。在`Queue2`中，类型 αt 是一个数据类型，并且`empty`和`enq`都是构造子。在`Queue3`中，同样 αt 是一个数据类型，结构还定义了签名`QUEUE`所要求的所有的值，以及额外的项目：`Queue`和`norm`。签名的实例可以包含签名中没有描述到的项目。

7.5 签名约束

结构的具有不同抽象程度的多个视图，可以通过使用不同的签名得到。结构可以在首次定义的时候被约束到签名上，也可以在稍后进行。约束既可以是透明的也可以是不透明的。

透明签名约束。我们到目前为止所使用的约束，都是由冒号（:）表示的，是透明的。要明白这意味着什么，让我们用签名`QUEUE`来约束现有的结构：

```
structure S1: QUEUE = Queue1;
structure S2: QUEUE = Queue2;
structure S3: QUEUE = Queue3;
```

这些声明使得`S1`、`S2`和`S3`相应地表示了与`Queue1`、`Queue2`和`Queue3`同样的结构。然而，新的结构是由签名`QUEUE`来约束的。类型 $\alpha \text{ Queue2.t}$ 和 $\alpha \text{ S2.t}$ 是一样的，但是`Queue2.empty`是一

个构造子，而`S2.empty`只能被用作一个值。结构`Queue3`和`S3`是相等的，但是`Queue3.norm`是一个函数，而`S3.norm`则什么也不是。

透明的签名约束可以隐藏组件，但是它们还是存在的，这不能被称作抽象。结构`S1`根本就没有隐藏它的表示；类型 α `S1.t`仍旧相当于 α `list`。

```
S1.deq ["We", "band", "of", "brothers"];
> ["band", "of", "brothers"] : string S1.t
```

结构`S2`和`S3`看上去可能更抽象些，因为它们声明了数据类型 α `t`，同时隐藏了它们的构造子。没有构造子就不能利用模式匹配来将类型的值拆散以暴露表示方法。然而，构造子`Queue3.Queue`却可以用在模式中来拆散类型 α `S3.t`的值：

```
val Queue3.Queue (heads, tails) =
  S3.enq (S3.empty, "Saint"), "Crispin");
> val heads = ["Saint"] : string list
> val tails = ["Crispin"] : string list
```

具体的结构`Queue3`，为它的抽象视图`S3`，提供了一个漏洞。

数据抽象在另一个方面也被危及。对于我们的每个队列结构来说，类型 α `t`都是允许相等测试的。相等测试比较的是内部表示，并不是队列。在表示法3下，值`([1,2], [])`和`([1], [2])`表示了同样的队列，而相等测试却说它们是不同的。

不透明签名约束。用符号`>`取代冒号可以使约束变成不透明的。这种约束隐藏了新结构中除了签名以外的所有信息。让我们通过约束具体的结构来建立真正抽象的队列结构：

```
structure AbsQueue1 :> QUEUE = Queue1;
structure AbsQueue2 :> QUEUE = Queue2;
structure AbsQueue3 :> QUEUE = Queue3;
```

受约束的结构组件是从原来结构的相应部分中分离出来的。结构`AbsQueue1`通过表来表示队列，但是我们看不到这点了：

```
AbsQueue1.deq ["We", "band", "of", "brothers"];
> Error: Type conflict:...
```

类似地，类型检测可以防止利用构造子`Queue3.Queue`来拆散结构`AbsQueue3`中的队列。相等测试也被禁止了：

```
AbsQueue3.empty = AbsQueue3.empty;
> Error: type 'a AbsQueue3.t must be an equality type
```

将类型指定为`eqtype` `t`而不是`type` `t`表示该类型允许相等测试。在签名中使用`eqtype`可以输出类型的相等测试，即使是在不透明约束中。

局限。不透明签名约束对于声明队列的抽象类型来说是非常合适的。抽象结构可以从现有的具体结构中得到，就像上面的`AbsQueue`声明那样，或者我们可以直接约束原始的结构声明：

```
structure Queue :> QUEUE = struct ... end;
```

不过，这两种签名约束给我们提供的是非此即彼的极端选择，这对于复杂的抽象类型来说很难使用。签名`DICTIONARY`指定了两个类型：`key`是搜索键值类型； α `t`是字典类型（4.14节）。

类型 α 应该是抽象的, 而 key 却应该是某种具体的类型, 例如字符串。否则的话, 我们就没有办法来引用键值了, 于是也没法调用`lookup`和`update`了! 下一节将介绍一种更为灵活的办法来声明抽象类型。

练习 7.6 基于表示法3, 说明只用抽象队列所提供的操作, 怎样构造内部表示不一样的两个相等的队列。

练习 7.7 扩展签名`QUEUE`, 增加函数`length`来返回队列中元素的个数, 以及`equal`来测试两个队列是否包含同样的元素序列。在结构`Queue1`、`Queue2`和`Queue3`中加入这些函数的声明。

7.6 抽象类型 (abstype) 声明

Standard ML有一种专门为抽象类型而设的声明形式。它完全隐藏了表示方法, 包括相等测试。`abstype`声明源自第一个ML版本, 它体现了早期结构化程序设计学派的思想。现在看来它显然是过时了。不过, 它比不透明约束更具选择性: 它可以有选择地应用在某个类型上, 而不是整个签名上。

简单的`abstype`声明包含两个部分, 数据类型绑定`DB`和声明`D`:

```
abstype DB with D end
```

数据类型绑定是一个后面跟着构造子描述的类型名字, 和出现在`datatype`声明中的完全一样。构造子在声明部分`D`中是可见的, 那里必须使用它们来实现与抽象类型相关的所有操作。在`D`中声明的标识符和类型名对外都是可见的, 但是类型的构造子被隐藏了。此外, 这个类型也不允许进行相等测试。

为了说明`abstype`声明, 让我们将它应用到队列上。声明本来应该包含在一个结构中以避免与内置的表函数`null`和`hd`发生名字冲突。但是这样的结构会使例子变得复杂, 我们就简单地在此另行命名。为了节省空间, 省略了异常。

作为表的队列。我们从表示法1开始。虽然`list`已经是一个数据类型, 但是`abstype`声明迫使我们在所有队列操作中使用新的构造子 (`Q1`)。这个构造子传统上称为抽象函数 (abstraction function), 它将具体表示映射到抽象值。

```
abstype 'a queue1 = Q1 of 'a list
  with
    val empty = Q1 [];

    fun enq(Q1 q, x) = Q1 (q @ [x]);

    fun qnull(Q1(x::q)) = false
      | qnull _         = true;

    fun qhd(Q1(x::q)) = x;

    fun deq(Q1(x::q)) = Q1 q;
  end;
```

作为回应, ML打出了声明过的标识符名字和类型:

```
> type 'a queue1
> val empty = - : 'a queue1
> val enq = fn : 'a queue1 * 'a -> 'a queue1
> val qnull = fn : 'a queue1 -> bool
```

```
> val qhd = fn : 'a queue1 -> 'a
> val deq = fn : 'a queue1 -> 'a queue1
```

267

`abstype`声明已经把`queue1`和`list`的联系隐藏起来了。

作为新数据类型的队列。现在转向表示法2。之前我们把构造子用小写字母命名为`empty`和`enq`，主要是为了使它们在外部可以作为值来使用，但那样是不规范的。现在`abstype`隐藏了构造子，我们又可以使用大写开头名字`Empty`和`Enq`了，因为终究还必须显式地输出它们的值：

```
abstype 'a queue2 = Empty
                    | Enq of 'a queue2 * 'a
with
val empty = Empty
and enq   = Enq

fun qnull (Enq _) = false
  | qnull Empty  = true;

fun qhd (Enq (Empty, x)) = x
  | qhd (Enq (q, x))    = qhd q;

fun deq (Enq (Empty, x)) = Empty
  | deq (Enq (q, x))    = Enq (deq q, x);
end;
```

我们不需要声明新的构造子`Q2`了，因为这个表示法本身需要它自己的构造子。除了队列类型的名字外，ML对这个声明的回应和对声明`queue1`的回应完全一样。外部的使用者只能通过输出的操作来处理队列。

这两个例子说明了`abstype`的主要特点。我们不需要再看`queue3`的类似声明了。

ML中的抽象类型：小结。ML对待抽象类型的处理并没有人们想像的那么直接，不过它也可以被简化成几个步骤。如果你想声明一个类型 t ，并只允许通过你选择的操作对其进行访问，那么下面告诉你如何进行。

1. 考虑一下是否输出 t 的相等测试。这只有当它的表示允许相等测试时才是适合的，并且这个相等测试要和抽象值的相等测试一致才行。另外也要考虑相等测试实际上是否必要。对于小对象，如日期和有理数，进行相等测试是适当的，但对于矩阵和弹性数组则不然。

268

2. 声明一个签名`SIG`来描述抽象类型和它的操作。如果类型允许相等测试，那么签名必须指定 t 为`eqtype`，否则就使用`type`。

3. 决定对`SIG`使用哪一种签名约束。不透明约束只有当签名中的所有类型都是抽象类型时才适合。

4. 书写结构（或函子）声明的框架，和上一步选择的约束方式联系起来。

5. 在`struct`和`end`框住的块内声明类型 t 和所需的操作。如果你使用透明签名约束，则它必须是一个`datatype`数据类型声明（以输出相等测试）或一个`abstype`抽象类型声明（以隐藏相等测试）。

`datatype`声明也可以产生抽象类型，这是因为签名约束隐藏了构造子。`abstype`或`datatype`声明建立的是全新的类型，ML认为不同于所有其他的类型。

函子`Dictionary`例证了第一个方案，而环形缓冲区结构`RingBuf`则例证了第二个方案。

练习 7.8 早期关于抽象类型的文章都提到了同一个例子：堆栈。有关的操作包括了`push`（将一项放在栈顶），`top`（返回栈顶项）以及`pop`（丢弃栈顶项）。除此之外，至少还需要另外两个操作。完成这个设计并使用`abstype`编写两个不同的实现。

练习 7.9 根据练习2.25为有理数书写一个`abstype`声明。使用`local`声明来隐藏任何辅助的函数。然后修改你的解决方案以得到符合签名`ARITH`的结构。

练习 7.10 为日期类型`date`设计并编写一个`abstype`声明，它用月和日来表示日期。（假设本年不是闰年。）提供函数`today`来将月和日转换成一个日期。提供函数`tomorrow`（明天）和`yesterday`（昨天），如果求得的日期在本年之外，它们应该抛出异常。

7.7 从结构导出的签名

结构声明可以没有签名约束，就像在`Queue1`、`Queue2`和`Queue3`中的签名那样。这种情况下，ML将自动导出一个完整描述结构内部细节的签名。

签名`QUEUE1`等价于为结构`Queue1`而导出的签名。它将`t`指定为一个`eqtype`，即允许相等测试的类型，因为表是可以进行相等比较的。可以看到，值的类型使用了类型 `α list`，而不是`QUEUE`中的 `αt` 。

```
signature QUEUE1 =
  sig
    eqtype 'a t
    exception E
    val empty : 'a list
    val enq   : 'a list * 'a -> 'a list
    val null  : 'a list -> bool
    val hd    : 'a list -> 'a
    val deq   : 'a list -> 'a list
  end;
```

为`Queue2`导出的签名将 `αt` 描述为一个数据类型，并具有构造子`empty`和`enq`，构造子不再被描述成值。这个签名可以如下声明：

```
signature QUEUE2 =
  sig
    datatype 'a t = empty | enq of 'a t * 'a
    exception E
    val null : 'a t -> bool
    val hd   : 'a t -> 'a
    val deq  : 'a t -> 'a t
  end;
```

为结构`Queue3`导出的签名同样将 `αt` 描述为一个数据类型，而不只是`QUEUE`中的类型。签名中描述了所有结构中的项目，包括了`Queue`和`norm`。

```
signature QUEUE3 =
  sig
    datatype 'a t = Queue of 'a list * 'a list
    exception E
    val empty : 'a t
    val enq   : 'a t * 'a -> 'a t
```

```

val null  : 'a t -> bool
val hd    : 'a t -> 'a
val deg   : 'a t -> 'a t
val norm  : 'a t -> 'a t
end;

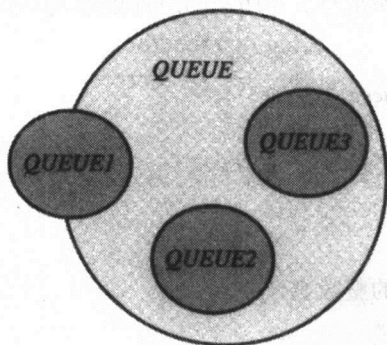
```

这些签名比 *QUEUE* 更具体、更明确。没有结构可以同时满足其中的多个签名。可以考虑 *QUEUE1* 和 *QUEUE3*。函数 *hd* 必须具有类型 $\alpha \text{ list}$ 以满足 *QUEUE1*；又必须具有类型 $\alpha \text{ t}$ 以满足 *QUEUE3*，后者还将 $\alpha \text{ t}$ 描述为数据类型，显然和 $\alpha \text{ list}$ 不同。

270

另一方面，每个签名可以有很多不同的实例。一个结构可以通过声明 x 为任何 *int* 类型值来满足描述 `val x : int`。它可以通过声明任何类型 t 来满足 `type t` 的描述。（但是，它只能通过完全相同的 `datatype` 声明来满足一个数据类型描述。）结构还可以包含签名中没有描述的项目。由此可见一个签名定义了一类结构。

这些类之间存在着有趣的关系。我们已经知道 *QUEUE1*、*QUEUE2* 和 *QUEUE3* 互不相交。后两个包含在 *QUEUE* 中；一个 *QUEUE2* 或 *QUEUE3* 的实例也是 *QUEUE* 的实例。一个 *QUEUE1* 的实例只有当它令类型 $\alpha \text{ t}$ 等价于 $\alpha \text{ list}$ 时才能成为 *QUEUE* 的实例。这些包含关系显示在下面的维恩图中：



练习 7.11 声明一个结构使其具有签名 *QUEUE1*，并以不同于 *Queue1* 的表示方法实现队列。

练习 7.12 声明一个结构使其具有签名 *QUEUE*，但是不需要实现队列。毕竟签名只描述了队列操作的类型，而不是它们的性质。

函子

ML 的函数是具有参数的表达式。对它的应用是将形式参数替换成实际参数值，然后返回所得到的表达式的值。函数只能应用在具有正确类型的参数上。

我们有几种队列的实现。能否书写一种使用队列，但又独立于任何特定队列实现的代码呢？这好像需要将结构作为参数。

271

函数本身也可以作为参数，因为在 ML 中函数也是值的一种。记录也是值，它们有点像结构，但是不能表示队列的实现，因为记录里面不能包含类型和异常构造子作为分量。

ML 的函子 (functor) 是一个以其他结构作为参数的结构。对它的应用是将形式参数替换为实际的结构参数，然后返回结果结构中出现的绑定。函子只能应用到具有正确签名的结构参数上。

函子允许我们书写以不同方式组合的程序单元。一个替换单元可以很快地连接进来，得到的新系统也可以很快地进行测试。函子也可以表达泛型算法。让我们看看这是怎样做的。

7.8 测试多个队列结构

下面是一个简单的队列测试框架。给定一个队列结构，它返回一个测试结构，里面含有两个函数。一个将表转换为队列；另一个则是前一个的逆操作。测试框架声明为一个函子，具有签名为`QUEUE`的结构参数：

```
functor TestQueue (Q: QUEUE) =
  struct
    fun fromList l = foldl (fn (x,q) => Q.enq(q,x)) Q.empty l;

    fun toList q = if Q.null q then []
                  else Q.hd q :: toList (Q.deq q);

  end;
> functor TestQueue : <sig>
```

函子内并没有引用现有的队列结构，而是引用参数`Q`。两个函数一致使用了队列操作。任何队列结构都可以进行测试，并测量它的效率。让我们从`Queue3`开始。将函子应用到这个参数上得到一个新的结构，我们称为`TestQ3`。`TestQ3`的组件就是测试`Queue3`的函数，这也可以从它们的类型中看到：

```
structure TestQ3 = TestQueue (Queue3);
> structure TestQ3 :
>   sig
>     val fromList : 'a list -> 'a Queue3.t
>     val toList   : 'a Queue3.t -> 'a list
>   end
```

测试用的数据就是从1到10 000的整数表：

```
val ns = upto(1,10000);
> val ns = [1, 2, 3, 4, ...] : int list
val q3 = TestQ3.fromList ns;
> val q3 = Queue ([1], [10000, 9999, 9998, 9997, ...])
> : int Queue3.t
val l3 = TestQ3.toList q3;
> val l3 = [1, 2, 3, 4, ...] : int list
l3 = ns;
> true : bool
```

`Queue3`通过了它的第一个测试：我们取回了原来的表。它的效率很高，只用了10毫秒就建立了`q3`，又用了50毫秒来将它转回到一个表。

ML对于`q3`声明的回应暴露了它使用一对表来表示队列的方法：`Queue3`没有定义抽象类型。应该试试结构`AbsQueue3`。同样，我们应用该函子并给结果结构命名：

```
structure TestAQ3 = TestQueue (AbsQueue3);
> structure TestAQ3 :
>   sig
>     val fromList : 'a list -> 'a AbsQueue3.t
>     val toList   : 'a AbsQueue3.t -> 'a list
>   end
```

```
val q = TestAQ3.fromList ns;
> val q = - : int AbsQueue3.t
```

这次ML没有暴露任何表示方法。在效率方面，*Queue3*和*AbsQueue3*没有区别。类似的评测显示*AbsQueue3*比*Queue1*和*Queue2*快很多倍，比练习7.4所建议的平衡树实现也快不少。由于*Queue1*是用表来表示队列的，它可以专门实现高效的*fromList*和*toList*，但是在函子内，我们只能使用签名*QUEUE*中所指定的操作。

更为实际的测试将涉及到队列的应用，例如广度优先搜索。函数*breadthFirst* (5.17节)为了简单起见使用了表来代替队列。使用函子可以独立地表达这个搜索策略，而不依赖队列的实现。

```
functor BreadthFirst (Q: QUEUE) =
  struct
    fun enlist q xs = foldl (fn (x,q) => Q.enq(q,x)) q xs;
    fun search next x =
      let fun bfs q =
          if Q.null q then Nil else
            let val y = Q.hd q
              in Cons(y, fn()=> bfs (enlist (Q.deq q) (next y)))
            end
          in bfs (Q.enq(Q.empty, x)) end;
      end;
  end
> functor BreadthFirst : <sig>
```

函数*enlist*将整个表的元素追加到队列尾部。让我们将这个函子应用到一个高效的队列结构上：

```
structure Breadth = BreadthFirst (Queue3);
> structure Breadth :
> sig
>   val enlist : 'a Queue3.t -> 'a list -> 'a Queue3.t
>   val search : ('a -> 'a list) -> 'a -> 'a seq
> end
```

函数*Breadth.search*等价于*breadthFirst*，不过要快得多。

很多语言没有和函子类似的机制。C语言通过使用头文件和包含文件来取得类似的效果。像这样的原始方法虽然大有帮助，但是它们不能处理错误。包含错误的文件导致错误的代码被编译：我们会得到很多层的错误信息。如果函子应用到了错误的结构上又会怎样呢？试试将*BreadthFirst*应用到标准库结构*List*上：

```
structure Wrong = BreadthFirst (List);
> Error: unmatched type spec: t
> Error: unmatched exception spec: E
> Error: unmatched val spec: empty
> Error: unmatched val spec: enq
> Error: unmatched val spec: deq
```

我们可以得到确切的错误信息，它描述了参数里面缺少了什么。上面没有提示缺少*hd*和*null*，因为*List*里面有同名的组件。

将队列结构作为参数所造成的复杂可能是没有必要的。*AbsQueue3*是最好的队列结构，我们可以将它改名为*Queue*来直接使用，就像使用标准库里的结构，如*List*，一样。但是通常我

们是有选择的。存在着多个可能的字典和优先队列的表示法。即便在标准库中也有多个各有千秋的结构来处理实数运算。在考虑泛型操作的时候，函子的作用是不容置疑的。

练习 7.13 考虑在其他你了解的语言中怎样获得ML模块的效果。你会怎样表达类似*QUEUE*的签名，可替换结构*Queue1*和*Queue2*，以及函子*TestQueue*？

练习 7.14 在什么范围内，*TestQueue*是测试队列的好工具？

7.9 泛型矩阵运算

有关联的结构除了性能以外还有其他不同点。2.22节中我们讨论过签名*ARITH*，它描述了*zero*、*sum*、*diff*、*prod*等几个组件。适合这个签名的实例包括了实现整数、实数、复数和有理数上的运算结构。第3章进一步提到了更多的可能性：二进制数、矩阵和多项式。

为了说明函子，让我们为矩阵运算编写一个泛型结构。为了简单起见，我们只考虑零、求和以及乘积：

```
signature ZSP =
  sig
    type t
    val zero : t
    val sum : t * t -> t
    val prod : t * t -> t
  end;
```

我们将声明一个函子，它的参数和结果结构都符合签名*ZSP*。

声明矩阵函子。已知一个类型*t*和它的三种算术操作，函子*MatrixZSP*声明了*t*上的矩阵类型和相应的矩阵运算（图7-1）。在研究这个函子体之前，不妨复习一下3.10节。

```
functor MatrixZSP (Z: ZSP) : ZSP =
  struct
    type t = Z.t list list;

    val zero = [];

    fun sum (rowsA, []) = rowsA
      | sum ([], rowsB) = rowsB
      | sum (rowsA, rowsB) = ListPair.map (ListPair.map Z.sum)
                                      (rowsA, rowsB);

    fun dotprod pairs = foldl Z.sum Z.zero (ListPair.map Z.prod pairs);

    fun transp ([]::_ ) = []
      | transp rows = map hd rows :: transp (map tl rows);

    fun prod (rowsA, []) = []
      | prod (rowsA, rowsB) =
        let val colsB = transp rowsB
        in map (fn row => map (fn col => dotprod(row, col))
                    colsB)
            rowsA
        end;

    end;
```

图7-1 泛型矩阵运算的函子

在函子头中，第二次出现的ZSP是返回结构的签名约束。因为这个约束是透明的，所以MatrixZSP不会返回抽象类型。假如使用不透明约束，那么就只能通过输出的零、求和以及乘积来对矩阵进行操作：我们将只能表达零矩阵！而现在的写法，允许将矩阵写成表的表。

返回的结构根据矩阵的元素类型Z.t声明了矩阵的类型t。这个声明是结果签名所要求的，签名里面指定了类型t。在函子体中并没有使用它。

然后，结构里声明了zero。在代数中，任何 $m \times n$ 的全零元素矩阵都叫做零矩阵（zero matrix）。签名ZSP中描述的zero : t要求我们声明唯一的一个零元素。因此，函子中将zero声明为空表，并使得sum和prod满足定律 $0 + A = A + 0 = A$ 以及 $0 \times A = A \times 0 = 0$ 。

结构声明了函数sum，用于计算两个矩阵相加。通过将行中对应元素相加来将两行相加，这里使用了库函数ListPair.map。类似地，两个矩阵相加是通过对应行的相加来完成的。矩阵的加法sum和矩阵元素的加法Z.sum之间并无冲突。

结构中的其他函数是为了辅助乘积函数prod的声明而设。点积的计算也是通过ListPair.map串起来的，而矩阵转置则是像5.7节中那样声明的。由于transp不能处理空表，因此函数prod要处理这种特殊情形。

因为ListPair中的函数都丢弃了无法匹配的多余表元素，所以这里没有检测矩阵的大小。于是，将 2×5 的矩阵和 3×4 的矩阵相加会得到一个 2×4 的矩阵，而不会抛出异常。

数值应用。在应用上面这个函子之前，必须先创建一些结构。我们已经看过实数矩阵了，现在轮到整数矩阵了。结构IntZSP恰好包括ZSP所描述的那些操作：

```
structure IntZSP =
  struct
    type t = int;
    val zero = 0;
    fun sum (x,y) = x+y: t;
    fun prod (x,y) = x*y: t;
  end;
> structure IntZSP :
>   sig
>     eqtype t
>     val prod : int * int -> t
>     val sum  : int * int -> t
>     val zero : int
>   end
```

将上面的函子应用到IntZSP之上可以为整数矩阵上的运算建立一个结构。这里举两个例子，

分别是求和 $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 6 & 8 \\ 10 & 12 \end{pmatrix}$ 以及乘积 $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 4 & 3 \end{pmatrix}$ 。

```
structure IntMatrix = MatrixZSP (IntZSP);
> structure IntMatrix : ZSP
IntMatrix.sum ([[1,2],[3,4]], [[5,6],[7,8]]);
> [[6, 8], [10, 12]] : IntMatrix.t
IntMatrix.prod ([[1,2],[3,4]], [[0,1],[1,0]]);
> [[2, 1], [4, 3]] : IntMatrix.t
```

2.21节中声明的结构Complex里的一些组件并没有在ZSP中进行描述。不过，签名匹配会忽略额外的组件。因此，我们可以将这个结构作为MatrixZSP的参数，产生的结果是复数矩阵运算

277 的结构。

```
structure ComplexMatrix = MatrixZSP (Complex);
> structure ComplexMatrix : ZSP
```

将一个结构运用于多种目的的技巧是保持程序简洁的有力工具。首要的是，这需要精心设计的签名。一致的命名习惯可以帮助保证不同的模块间彼此配合。

图论应用。组件`zero`、`sum`和`prod`并不一定要有明显的数值解释。许多图论算法都是在矩阵上操作的，这些矩阵对（关于元素的）0、+和×有出人意料的解释。

由 n 个结点组成的有向图可以用 $n \times n$ 邻接矩阵（adjacency matrix）来表示。矩阵元素 (i, j) 是一个布尔值，代表是否存在一条由结点 i 到结点 j 的边。典型的矩阵操作将元素`zero`解释成假，而`sum`是析取，`prod`是合取。

```
structure BoolZSP =
  struct
    type t = bool;
    val zero = false;
    fun sum (x,y) = x orelse y;
    fun prod (x,y) = x andalso y;
  end;
> structure BoolZSP :
> sig
>   eqtype t = bool
>   val prod : bool * bool -> bool
>   val sum  : bool * bool -> bool
>   val zero : bool
> end
```

若 A 是一个布尔邻接矩阵，并且由 A 给出的图中从 i 到 j 恰有一条长度为2的路径，则 $A \times A$ 就表示了这样一个图：它有一条从 i 到 j 的边。矩阵运算可以计算出图的传递闭包。然而，位操作（在标准库的结构`Word8`中）可以更快地进行这样的计算，因此，让我们转而看看另一个更非同寻常的例子。

将`zero`定义为无穷大（ ∞ ）、`sum`定义为最小值（ \min ）以及将`prod`定义为和（+）。除去无穷以外的其他操作对于无穷大进行了如下推广： $\min(\infty, x) = \min(x, \infty) = x$ 以及 $\infty + x = x + \infty = \infty$ 。这样一来，三元组 $(\infty, \min, +)$ 多少满足了与 $(0, +, \times)$ 同样的定律。然而，这样奇怪的算术运算又有什么意义呢？

278

考虑这样一个有向图，它的边被标记上数值，以表示沿该边通过的代价。（有可能是负数！）相应的邻接矩阵的元素就是一些数值。元素 (i, j) 就是从 i 到 j 的边的代价，或者如果是无穷的话则表示没有这条边。设 A 是一个邻接矩阵，并使用上面奇特的算术运算来计算 $A \times A$ 。乘积的元素 (i, j) 是从 i 到 j 长度为2的诸路径的最小代价。我们有了必需的方法，用它可以表达这个计算图中所有结点间最短路径的标准算法。

下面就是实现这个奇特运算的结构。它是基于`int`类型的。它并没有将`zero`声明为无穷大，而是一个很大的整数。^① 它将`sum`声明为标准库中的最小值函数，并且将`prod`声明为加法的推广。

① 标准库中`Int`结构的组件`maxInt`要么形如`SOME n`，其中 n 是可以表示的最大整数；要么形如`NONE`。任何超过所有边标记的绝对值和的整数都是适用的。

```

structure PathZSP =
  struct
    type t = int;
    val SOME zero = Int.maxInt;
    val sum       = Int.min
    fun prod(m,n) = if m=zero orelse n=zero then zero
                     else m+n;
  end;

```

将我们的函子应用到这个结构上便生成了奇特的矩阵运算结构。“所有结点间最短路径”的算法可以仅用几行程序编写：

```

structure PathMatrix = MatrixZSP (PathZSP);
> structure PathMatrix : ZSP

fun fast_paths mat =
  let val n = length mat
      fun f (m,mat) = if n-1 <= m then mat
                      else f(2*m, PathMatrix.prod(mat,mat))
      in f (1, mat) end;
> val fast_paths = fn : PathMatrix.t -> PathMatrix.t

```

Cormen等(1990)讨论了这个算法(26.1节)。我们来试一下其中的一个例子。给定一个五结点的邻接矩阵，`fast_paths`返回预期的结果：

```

val zz = PathZSP.zero;
val zz = > 1073741823 : int
fast_paths [[0, 3, 8, zz, ~4],
            [zz, 0, zz, 1, 7],
            [zz, 4, 0, zz, zz],
            [2, zz, ~5, 0, zz],
            [zz, zz, zz, 6, 0]];
> [[0, 1, ~3, 2, ~4],
> [3, 0, ~4, 1, ~1],
> [7, 4, 0, 5, 3],
> [2, ~1, ~5, 0, ~2],
> [8, 5, 1, 6, 0]] : PathMatrix.t

```

279

函子`MatrixZSP`的参数是仅有四个组件的结构。下节将会看到，甚至更小的结构也都有它们的用途。

❶ 一个代数的视角。Cormen等(1990)更进一步地给这个奇特的运算建立了完整的基础。他们(26.4节)定义了闭合半环(closed semiring)的记法，并讲述了它和路径算法的联系。闭合半环涉及了类似于0、1、+和×的操作，它们满足一系列代数定律：+和×要满足交换律和结合律等等。关于闭合半环的签名需要在`ZSP`上补充一个组件：`one` (一)。ML的模块对于将这样的抽象付诸使用来说是非常理想的。

练习 7.15 声明另一版本的`PathZSP`，其中将 ∞ 表示为特殊的值，而不等于任一个整数。这样的结构适用于类似Poly/ML这样的ML系统，在这些系统中，类型`int`没有最大值。

练习 7.16 矩阵并不一定要是表的表。研究标准库中的向量结构`Vector`，然后书写一个函子`VMatrixZSP`，将矩阵表示为向量的向量。

7.10 泛型的字典和优先队列

在第4章，我们实现了字符串上的二叉搜索树，以及实数上的优先队列。我们可以利用函子来取消类型上的限制，将这两个数据结构推广到任意的有序类型。有序类型以及它的比序函数将包装在含有两个组件的结构中。

排序也可以类似地进行推广——并不需要函子。只需简单地将比序函数作为参数传入，将排序表示成一个高阶函数。但是，这种办法之所以可行，仅仅是由于排序是一种一体化的操作。将优先队列的那些操作变为高阶函数会使得一些荒谬的做法成为可能，例如用一种顺序插入项目，而用另一种顺序删除项目。

有序类型作为结构。数学家将有序集定义为一个序偶 $(A, <)$ ，其中 A 是集合， $<$ 是 A 上满足传递等性质的关系。ML的模块可以表示这样的数学概念，尽管记法比较难看。签名`ORDER`描述了一个类型 t 和一个比序函数`compare`：

280

```
signature ORDER =
  sig
    type t
    val compare: t*t -> order
  end;
```

回忆一下，ML库将`order`声明为一个枚举类型，它具有三个构造子：`LESS`、`EQUAL`和`GREATER`。像`String`、`Int`和`Real`这样的库结构都含有组件`compare`，这个函数有两个同样的相应类型的操作数。举例来说，我们可以这样包装字符串比序：

```
structure StringOrder: ORDER =
  struct
    type t = string;
    val compare = String.compare
  end;
> structure StringOrder : ORDER
```

我们可以定义自己的比序函数，但是需要注意，二叉搜索树要求序是线性的（linear）。一个序 $<$ 是线性的当它满足：对于所有的 x 和 y ， $x < y$ 、 $x = y$ 和 $x > y$ 三者有且仅有一个成立。在这里，它的意思是：如果比较的结果是`EQUAL`的话，那么两个操作数就确实是相等的。对于优先队列，我们可以使用偏序，也就是说，如果两项比较结果是`EQUAL`只是说明两者具有相同的优先级，即使它们本身是不同的。（但是，还请参见下面的练习7.23。）

字典函子。4.14节通过声明签名`DICTIONARY`来勾勒出字典的各项操作，并且使用二叉搜索树来实现。这个实现有两个方面的缺陷：一个是键值被人为地限制在类型`string`上，另一个是树的表示方法从结构外部是可见的。

新的实现方法是（图7-2）通过将有序结构作为参数来弥补第一个缺陷，第二个缺陷则是通过`abstype`声明的方式来弥补。这个实现禁止了相等测试，这是因为不同的二叉搜索树可能表示同样的字典。

函子头告诉我们，仅有的键值操作只是比较。函子体则类似于之前有缺陷的结构，只不过它使用了其参数`Key.compare`而不是`String.compare`来进行比较。并且，函子将键值类型`key`声明作`Key.t`，而在旧的结构中，这个类型是`string`。

将函子`Dictionary`应用到结构`StringOrder`上就建立了以字符串作为键值的字典结构。

```
structure StringDict = Dictionary (StringOrder);
> structure StringDict : DICTIONARY
```

281

```
functor Dictionary (Key: ORDER) : DICTIONARY =
  struct

    type key = Key.t;

    abstype 'a t = Leaf
      | Bran of key * 'a * 'a t * 'a t
    with

      exception E of key;

      val empty = Leaf;

      fun lookup (Leaf, b)          = raise E b
        | lookup (Bran(a,x,t1,t2), b) =
          (case Key.compare(a,b) of
             GREATER => lookup(t1, b)
          | EQUAL    => x
          | LESS     => lookup(t2, b));

      fun insert (Leaf, b, y)       = Bran(b, y, Leaf, Leaf)
        | insert (Bran(a,x,t1,t2), b, y) =
          (case Key.compare(a,b) of
             GREATER => Bran(a, x, insert(t1,b,y), t2)
          | EQUAL    => raise E b
          | LESS     => Bran(a, x, t1, insert(t2,b,y)));

      fun update (Leaf, b, y)       = Bran(b, y, Leaf, Leaf)
        | update (Bran(a,x,t1,t2), b, y) =
          (case Key.compare(a,b) of
             GREATER => Bran(a, x, update(t1,b,y), t2)
          | EQUAL    => Bran(a, y, t1, t2)
          | LESS     => Bran(a, x, t1, update(t2,b,y)));

    end

  end;
```

图7-2 作为二叉搜索树的字典函子

现在可以创建和搜索字典了。这里，声明一个中缀操作符可以消除难看的对于`update`的嵌套调用：

```
infix |> ;
fun (d |> (k,x)) = StringDict.update(d,k,x);

val dict = StringDict.empty
  |> ("Crecy",1346)
  |> ("Poitiers",1356)
  |> ("Agincourt",1415)
  |> ("Trafalgar",1805)
```

```

|> ("Waterloo",1815);
> val dict = - : int StringDict.t
StringDict.lookup(dict,"Poitiers");
> 1356 : int

```

优先队列：一个子结构的例子。4.16节通过声明签名`PRIORITY_QUEUE`勾勒了优先队列的各个操作，并使用二叉树来实现它。该实现同样具有字典的那两个缺陷。与其再阐述一遍相同的思想，不如让我们来看一些新东西：子结构。

字典和优先队列的一个区别在于有序所扮演的角色。字典函子以有序结构作为参数是因为它使用了搜索树，另一种实现可能是以相等测试或散列函数作为参数。然而，优先队列在本质上是关于有序的：在积累了一定数量的项目后，它首先返回最小的那一项。因此，我们来修改一下最后的签名，来显式地描述有序结构：

```

signature PRIORITY_QUEUE =
  sig
    structure Item : ORDER
    type t
    val empty      : t
    val null       : t -> bool
    val insert     : Item.t * t -> t
    val min        : t -> Item.t
    val delmin     : t -> t
    val fromList   : Item.t list -> t
    val toList     : t -> Item.t list
    val sort       : Item.t list -> Item.t list
  end;

```

签名`PRIORITY_QUEUE`中描述了一个子结构`Item`，它与签名`ORDER`匹配。项的类型是`Item.t`，而优先队列的类型则是`t`。因此，返回队列中最小项的函数`min`具有类型`t -> Item.t`。

所有优先队列结构都内含了有序结构。如果`PQueue`是该签名的一个实例，就可以通过书写

```
PQueue.Item.compare(x,y)
```

来比较`x`和`y`。在这一方案下，系统组件被描述为子结构。前一个版本的`PRIORITY_QUEUE`将项目描述为类型`item`，而不是结构`Item`，许多人都喜欢它的简单性。

相应的函子勾勒如下。函子体的大部分都忽略掉了，这部分类似于第4章的图4-4。

```

functor PriorityQueue (Item: ORDER) : PRIORITY_QUEUE =
  struct
    structure Item = Item;

    fun x <= y = (Item.compare(x,y) <> GREATER);

    abstype t = ...
      with
        :
      end
  end;

```

其中结构`Item`的声明看上去没什么用，因为`Item`在函子体中已经是可见的了，但是结果签名需

要这个声明。这与我们在结构和函子中所见过的类型声明类似。嵌套的结构声明不一定要非常简单，所有允许在顶层声明的形式都可以出现在另一个结构中。

上面的函子重新声明了中缀操作符`<=`，用以表示项上的“小于或等于”。在第4章，二叉树使用`compare`来进行比序，而优先队列则使用`<=`。为两个函子声明两种不同的签名是不合理的，同样也不应该描述所有的关系操作符。简单统一的接口将使模块更易于配合。

`abstype`声明可以使用全新的树构造子，就像在`Dictionary`中一样。或者也可以通过一个类似之前的抽象类型`queue1`中那样的哑构造子来使用已有的构造子`Lf`和`Br`（4.13节中声明在顶层）。

练习 7.17 书写一个新版本的函子`Dictionary`，将字典表示为序偶(`key`, `item`)的表，表中元素根据键值排序。

练习 7.18 完成上面的`abstype`声明，尝试所述的两种选择。你喜欢哪一种？

练习 7.19 书写一个新版本的函子`PriorityQueue`，将优先队列表示为一个升序表，而不是二叉树。

练习 7.20 书写一个函子`Sorting`，它以签名`ORDER`的实例作为参数，其结果结构要实现快速排序及合并排序。提供两种以上排序算法的出发点是什么？

282
284

利用模块建立大型系统

通过一些小例子，我们已经了解了模块语言的基本功能。我们已经看到过很多种结构的使用方法：

- 库结构`List`包含了相关的声明，但是，我们可以用表的构造子写出更多的声明。
- 结构`AbsQueue3`输出了一个抽象类型，以及它的所有基本操作。更多关于队列的操作只能用这些基本操作来表示。
- 满足签名`ZSP`的那些结构是作为一个函子的参数或结果来使用的。它们只有几个组件，也就是那些与函子相关的操作。

一个大型系统应该被组织成数百个如上所述的小结构。这种组织应该是有层次的：主要的子系统应该实现为结构，其中的组件是更为下层结构。比较混乱的程序员可能会发现自己在管辖着几个巨型结构，每个结构都是由成百上千的组件组成的。

组织得好的系统会有许多小的签名。组件的描述将遵守严格的命名约定。在小组项目中，组员必须在每个签名上取得一致，之后对于签名的改动一定要严格控制。

系统将包括一些，也可能是很多的函子。如果主要的子系统都是独立实现的，那么它们就都必须是函子。

模块语言包含了使所有这些行之有效的做法成为可能的构造，其中许多构造是晦涩的，因此让我们更进一步地看看模块。

7.11 多参数函子

一个ML函数只有一个参数。多个参数通常包装成一个元组。另外，它们也可以包装成一个记录。高阶函数可以通过柯里化的机制来表示多个参数。

一个函子也只有一个参数。多个参数包装成为一个结构，这和将函数的参数作为记录传

285

人类似。这方面的语法比较臃肿，不过却是可行的。有些编译器通过提供高阶函子对Standard ML进行了扩展，高阶函子允许柯里化。

字典顺序函子。第一个例子是带两个参数的函子。若 $<_\alpha$ 是类型 α 上的序关系，并且 $<_\beta$ 是类型 β 上的序关系，则类型 $\alpha \times \beta$ 上的字典序 (lexicographic ordering) $<_{\alpha \times \beta}$ 定义为

$$(a', b') <_{\alpha \times \beta} (a, b) \text{ 当且仅当 } a' <_\alpha a \text{ 或 } (a' = a \text{ 且 } b' <_\beta b)$$

函子`LexOrder`的结果签名是`ORDER`。它需要两个形式参数：结构`O1`和`O2`，也都具有签名`ORDER`。该函子的声明说明了ML函子头的一般语法：

```
functor LexOrder (structure O1: ORDER
                  structure O2: ORDER) : ORDER =
  struct
    type t = O1.t * O2.t;
    fun compare ((x1,y1), (x2,y2)) =
      (case O1.compare (x1,x2) of
        EQUAL => O2.compare (y1,y2)
       | ord   => ord)
  end;
```

形式参数表就是一个签名描述——没有括住两端的`sig`和`end`的一个签名。被描述的组件在函子体中是可见的。这个函子可以应用到任何匹配该描述的结构上：任何含有两个子结构的结构，这两个子结构需匹配签名`ORDER`。作为参数的结构可以通过任何结构表达式给出，也包括函子应用。

之前声明过结构`StringOrder`，也可以类似地声明结构`IntegerOrder`。我们可以提供这两个结构给函子：

```
structure StringIntOrd = LexOrder(structure O1=StringOrder
                                   structure O2=IntegerOrder);
> structure StringIntOrd : ORDER
```

由声明列表组成的实际参数被看作是一个结构表达式。这多个参数（声明）构成了结构体，并且我们可以省略括住两端的`struct`和`end`。

下面的演示将提醒我们这个函子的目的。将字符串的序关系和整数的序关系组合在一起产生了序偶（字符串，整数）上的序关系。字符串上的序优先于整数上的。

286

```
StringIntOrd.compare (("Edward", 3), ("Henry", 2));
> LESS : order
StringIntOrd.compare (("Henry", 6), ("Henry", 6));
> EQUAL : order
StringIntOrd.compare (("Henry", 6), ("Henry", 5));
> GREATER : order
```

关联表，`eqtype`描述。ML关于多个参数的函子语法并不要求那些参数一定是结构。它们可以是签名中所能够描述的任何东西，包括类型、值和异常。

接下来的例子演示了`eqtype`描述，以及一般的函子语法。之前，我们已经用二叉搜索树实现过字典，序偶表简单些，但比较慢。和3.16节中的一样，这里查找操作使用相等测试来比较键值。

`eqtype`描述可以出现在任何一个签名中，它描述了允许相等测试的类型。一个结构只有声明了真正允许相等测试的类型才能匹配这样的签名。在函子体中，可以在用`eqtype`描述的

类型上进行相等测试。

在例子中，函子的形式参数表是一个签名描述（见图7-3）。里面只有一个参数，也就是一个相等类型。一般的函子语法允许我们将`AssocList`看作是一个形式参数为一个类型的函子。因为类型`key`被描述为`eqtype`，所以它允许在`AssocList`内进行相等测试。下面是两个函子的应用：

```
structure StringIntAList = AssocList (type key = string*int);
> structure StringIntAList : DICTIONARY

structure FunctionAList = AssocList (type key = int->int);
> Error: type key must be an equality type
```

可以将函子应用到`string × int`上是因为这个类型允许相等测试，类型`int → int`则被拒绝了。

```
functor AssocList (eqtype key) : DICTIONARY =
  struct
    type key = key;
    type 'a t = (key * 'a) list;

    exception E of key;

    val empty = [];

    fun lookup ((a,x)::pairs, b) = if a=b then x
                                   else lookup(pairs, b)
      | lookup ([], b)           = raise E b;

    fun insert ((a,x)::pairs, b, y) = if a=b then raise E b
                                       else (a,x)::insert(pairs, b, y)
      | insert ([], b, y)          = [(b,y)];

    fun update (pairs, b, y) = (b,y)::pairs;

  end;
```

图7-3 使用关联表的字典函子

无参数函子。空结构（empty structure）没有任何组件：

```
struct end
```

它的签名是空签名（empty signature）：

```
sig end
```

空结构主要用作函子的实际参数。这类似于空元组`()`，它主要在函数不依赖其参数值时使用。可以回忆一下如何使用函数来表示序列尾（5.12节）。空参数也和命令式程序设计一起使用。这里，函子的例子涉及到了引用，我们将在第8章进行讨论。

函子`MakeCell`带有一个空参数。它的空形式参数表构成了一个空签名。每次调用`MakeCell`时，它都分配了一个新的引用单元，并将其作为结构的一部分返回。一开始这个单元等于0：

```
functor MakeCell () = struct val cell = ref 0 end;
> functor MakeCell : <sig>
```

下面是两个函子调用。空的实际参数表构成了空结构体。

```
structure C1 = MakeCell ()
and      C2 = MakeCell ();
> structure C1 : sig val cell : int ref end
> structure C2 : sig val cell : int ref end
```

结构C1和C2是以同样方式创建的，但是它们包含了不同的引用单元。我们将1存入C1单元，然后查看它们两个的情况：

```
C1.cell := 1;
> () : unit
C1.cell;
> ref 1 : int ref
C2.cell;
> ref 0 : int ref
```

两个单元存有不同的整数。因为MakeCell是函子而不是结构，所以它能够需要多少不同的单元就分配多少。

△ 函子语法的混淆。一般函子语法都在函子头中放置一个签名描述，由此可以处理任意数量的参数。但是，当恰好只有一个结构作为参数时又怎样呢？我们可以使用基本函子语法，它比一般语法更为简练直接，一般语法需要建立另一个结构。另一方面，同时在一个程序中使用两种语法又可能导致混淆。我们早期的所有例子都使用基本语法：

```
functor TestQueue (Q: QUEUE) ...
```

另外一个程序员有可能已经使用了一般语法：

```
functor TestQueue2 (structure Q: QUEUE) ...
```

这两个声明的区别仅在于形式参数表中关键字structure，有可能被忽视。要想避免错误信息，函子需要以相应的参数语法来调用：

```
TestQueue (Queue3)
TestQueue2 (structure Q = Queue3)
```

为统一起见，有些程序员倾向于只使用一般语法。

练习 7.21 书写另一版本的AssocList，其中不涉及eqtype。而使用类似ORDER的签名取而代之。

练习 7.22 函子AssocList并未隐藏字典的表示方法，书写另一版本，在其中声明一个抽象类型。

练习 7.23 在偏序关系中，有些元素对可以是不相关的。将这种情况记为EQUAL通常都不会令人满意，这样做在字典序的定义中会给出错误的结果。John Reppy建议用类型order option的值来表示比较结果，使用NONE来标记“无关系”。请仿照ORDER和LexOrder为偏序声明签名PORDER，并声明用于组合偏序关系的函子LexPOrder。

练习 7.24 （继续上面的练习。）若 α 是一类型， $<_{\beta}$ 是类型 β 上的偏序关系，并且 f 是具有类型 $\alpha \rightarrow \beta$ 的函数，则可以定义类型 α 上的偏序关系 $<$ ，满足 $x' < x$ 当且仅当 $f(x') <_{\beta} f(x)$ 。（注意，

$f(x') = f(x)$ 并不需要蕴涵 $x' = x$ 。)请声明一个三参数函子来实现这个定义。

练习 7.25 哪些结构是空签名的实例？换句话说，哪些结构可以作为函子`MakeCell`的合法参数？

7.12 共享约束

当模块被组合在一起形成更大的模块时，需要特别小心地去保证各组件彼此配合。考虑将字典和优先队列组合在一起的问题，这时要保证它们的类型是一致的。

上面，我们将函子`Dictionary`应用到参数`StringOrder`上，创建了结构`StringDict`。接着，我们把`dict`声明为一个以字符串为索引的字典。类似地，我们可以将`PriorityQueue`应用到`StringOrder`上，并创建字符串的优先队列结构。

```
structure StringPQueue = PriorityQueue (StringOrder);
> structure StringPQueue : PRIORITY_QUEUE
```

现在我们来将`pq`声明为字符串的优先队列：

```
StringPQueue.insert("Agincourt", StringPQueue.empty);
> - : StringPQueue.t
StringPQueue.insert("Crecy", it);
> - : StringPQueue.t
val pq = StringPQueue.insert("Poitiers", it);
> val pq = - : StringPQueue.t
```

由于`pq`中的元素是字符串，并且`dict`是以字符串为索引，因此`pq`的最小元可以作为搜索`dict`的键值。

```
StringDict.lookup(dict, StringPQueue.min pq);
> 1415 : int
```

我们已经把字典和优先队列放在一起使用了，不过只是针对类型`string`。将上面这个表达式推广到任意的有序类型则需要一个函子。在函子体中，表达式形如

```
Dict.lookup(dict, PQueue.min pq)
```

其中，`PQueue`和`Dict`是分别匹配签名`PRIORITY_QUEUE`和`DICTIONARY`的结构。但是，里面的类型一致吗？

290

$$PQueue.min : PQueue.t \rightarrow PQueue.Item.t$$

$$Dict.lookup : \alpha Dict.t \times Dict.key \rightarrow \alpha$$

对于`Dict.lookup`的调用只有在`PQueue.Item.t`和`Dict.key`是同种类型时才是允许的。保证这点的办法是让函子自行建立结构`PQueue`和`Dict`。下面的函子以一个有序类型作为参数，并把它提供给函子`PriorityQueue`和`Dictionary`。上面的表达式将作为函数`lookmin`的函数体出现。

```
functor Join1 (Order: ORDER) =
  struct
    structure PQueue = PriorityQueue (Order);
    structure Dict   = Dictionary   (Order);

    fun lookmin(dict, pq) = Dict.lookup(dict, PQueue.min pq);

  end;
```


通常都会用到一个函子调用另一个函子。但是，函子`Join1`并没有组合现有的结构：它创建了新的结构。这种做法会创建很多重复的结构。

我们的函子应该采用现有的结构`PQueue`和`Dict`，检查它们的类型是否兼容。共享约束（sharing constraint）能够强制类型一致：

```
functor Join2 (structure PQueue : PRIORITY_QUEUE
                  structure Dict   : DICTIONARY
                  sharing type PQueue.Item.t = Dict.key) =
  struct
    fun lookmin(dict, pq) = Dict.lookup(dict, PQueue.min pq);
  end;
```

我们回到了多参数函子语法，因为共享约束是签名描述的一种形式。在函子体内，该约束确保了那两个类型是相等的，因此，类型检测器会接受`lookmin`的声明。当函子应用到实际的结构上时，ML编译器将要求那两个类型确实是一样的。

为了演示这个函子，我们需要整数类型的优先队列和字典：

```
structure IntegerPQueue = PriorityQueue (IntegerOrder);
> structure IntegerPQueue : PRIORITY_QUEUE
structure IntegerDict = Dictionary (IntegerOrder);
> structure IntegerDict : DICTIONARY
```

291

两个基于字符串的结构可以组合在一起，两个基于整数的结构也可以。在两种情形中，函数`lookmin`采用了基于相同类型的优先队列和字典。

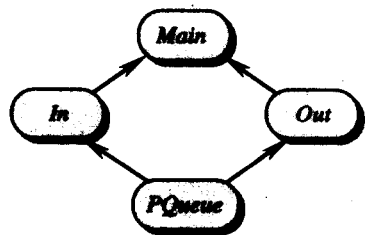
```
structure StringCom = Join2 (structure PQueue = StringPQueue
                                   structure Dict   = StringDict);
> structure StringCom
> : sig
>   val lookmin: 'a StringDict.t * StringPQueue.t -> 'a
> end

structure IntegerCom = Join2 (structure PQueue = IntegerPQueue
                                   structure Dict   = IntegerDict);
> structure IntegerCom
> : sig
>   val lookmin: 'a IntegerDict.t * IntegerPQueue.t -> 'a
> end
```

但是，如果我们试图混合两种类型的话，编译器会拒绝这个声明：

```
structure Bad = Join2 (structure PQueue = IntegerPQueue
                             structure Dict   = StringDict);
> Error: type sharing violation
> StringDict.key # IntegerPQueue.Item.t
```

结构上的共享约束。当函子组合系统组件时，其中的公共子结构可能需要共享约束。下面画的是一个典型的情况。结构`In`输入待解的问题，结构`Out`输出问题的解。这两个组件通过一个目标问题的优先队列进行通信，这个优先队列在结构`PQueue`中。结构`Main`通过`In`和`Out`协调这个程序。



假设`In`和`Out`匹配下面的两个签名:

```
signature IN =
  sig
    structure PQueue: PRIORITY-QUEUE
    type problem
    val goals: problem -> PQueue.t
  end;

signature OUT =
  sig
    structure PQueue: PRIORITY-QUEUE
    type solution
    val solve: PQueue.t -> solution
  end;
```

将`In`和`Out`组合在一起的函子大致是这样的:

```
functor MainFunctor (structure In: IN and Out: OUT
                      sharing In.PQueue = Out.PQueue) =
  struct
    fun tackle(p) = Out.solve(In.goals p)
  end;
```

因为结构`In.PQueue`和`Out.PQueue`是声明为共享的, 所以类型`In.PQueue.t`和`Out.PQueue.t`在函子体内是相同的。(留意`and`的使用, 它可以简洁地描述两个结构。)

在构造系统的时候, 需要将同一个结构`PQueue`放到`In`和`Out`中去。然后, 函子`MainFunctor`就会接受`In`和`Out`作为参数, 因为它们会满足共享约束。

理解共享约束。共享是ML模块中最难的方面之一。虽然共享约束可以出现在任何签名中, 但是它们仅在签名是描述函子参数时才是必需的。使用的函子越多, 需要的共享约束越多。

类型错误通常是警告哪里可能需要一个共享约束。在上一个例子中, 省略共享约束可能导致错误“类型冲突: 期望`In.PQueue.t`, 发现`Out.PQueue.t`。”不幸的是, 有些编译器会产生莫名其妙的错误信息。

类型错误可以通过在类型上强加一个共享约束来避免:

```
sharing type In.PQueue.t = Out.PQueue.t
```

实际用于`MainFunctor`中的结构共享约束要更强些: 它蕴涵的类型共享是一直进行到底的。它蕴涵的类型`In.PQueue.Item.t`和`Out.PQueue.Item.t`也是共享的。

ML通过比较类型的独立标识来强制共享约束。每个新的数据类型和抽象类型都被认为和之前所有的已有类型不同。

```
structure DT1 = struct datatype t = C end;
structure DT2 = struct datatype t = C end;
structure DT3 = struct type t = DT1.t end;
```

类型`DT1.t`和`DT2.t`是不同的, 尽管它们来自完全相同的`datatype`声明。类型缩写保持了类型的独立标识, 因此类型`DT1.t`和`DT3.t`是一样的。

练习 7.26 解释ML对于下列声明的回应。

```
signature TYPE = sig type t end;
functor Funny (structure A: TYPE and B: TYPE
```

```

        sharing A=B) = A;
structure S1 = Funny (structure A=DT1 and B=DT1);
structure S2 = Funny (structure A=DT2 and B=DT2);
structure S3 = Funny (structure A=S1 and B=S2);

```

练习 7.27 假设函子`Input`和`Output`是如下声明的:

```

functor Input (structure PQueue: PRIORITY_QUEUE): IN =
  struct
    structure PQueue = PQueue;
    fun goals ...;
  end;
functor Output (structure PQueue: PRIORITY_QUEUE): OUT =
  struct
    structure PQueue = PQueue;
    fun solve ...;
  end;

```

通过应用这些函子来声明可以提供给`MainFunctor`的结构。然后,声明具有所需签名但违反函子共享约束的结构。

练习 7.28 上面声明的函子`Input`和`Output`将形式参数`PQueue`合并到了结果结构中。修改它们以生成全新的`PRIORITY_QUEUE`的实例。这样将会如何影响`MainFunctor`?

7.13 全函子式程序设计

不言而喻,不应该声明一个只调用一次的过程。我们并未声明过只调用一次的函子。每个形式参数都可以在实际参数中进行选择,例如,参数`Order`可以被实例化成`StringOrder`或`IntegerOrder`。非泛型的程序单元则被编写成结构,而不是写成函子。

然而,声明过程现在被看作是良好的编程方式,即使它们仅被调用一次。有很好的理由去声明那些不仅仅是必需的函子。有些程序员几乎完全使用函子编写程序,只是在给函子提供参数时才书写结构。他们的函子和签名是自包含的:只引用其他签名以及标准库的组件。

如果所有的程序单元都编写成函子,那么就可以单独书写和编译它们。首先,声明签名,然后编写函子。当编译函子时,错误信息可以揭示签名中的缺失和错误。修正后的签名可以通过重新编译函子来检查。

函子可以以任意顺序进行编写。每个函子只引用签名,而不是结构或其他函子。一些人喜欢自顶向下编写,另一些则是自底向上。多个程序员可以独立地编写他们的函子。

一旦所有的函子书写和编译完毕,应用它们就能为每个程序单元生成一个结构。最后结构包含了可执行程序。一个函子可以被修改和重新编译,然后新的系统得以建立,如果签名没有改动,则不需要重新编译其他函子。应用函子相当于链接程序单元。可以为系统建立不同的配置。

二叉树函子。从4.13节开始,我们陆续声明了二叉树、弹性数组等结构。我们甚至将`tree`声明为顶层的数据类型。全函子方式要求每个程序单元都由一个自包含的签名来描述。

我们现在必须为二叉树声明一个签名。签名中必须描述数据类型`tree`,因为它将不在顶层声明。

```
signature TREE =
  sig
    datatype 'a tree = Lf | Br of 'a * 'a tree * 'a tree
    val size   : 'a tree -> int
    val depth  : 'a tree -> int
    val reflect : 'a tree -> 'a tree
  :
  end;
```

我们必须为Braun数组的操作声明一个签名。这个签名将Tree描述为一个子结构以提供对于类型tree的访问。[⊖]

295

```
signature BRAUN =
  sig
    structure Tree: TREE
    val sub      : 'a Tree.tree * int -> 'a
    val update   : 'a Tree.tree * int * 'a -> 'a Tree.tree
    val delete   : 'a Tree.tree * int -> 'a Tree.tree
  :
  end;
```

签名FLEXARRAY (4.15节) 是自包含的, 因为它只依赖于标准类型int。签名ORDER和PRIORITY_QUEUE (7.10节) 也是自包含的。由于一个签名可能引用其他签名, 因此必须以正确的顺序进行声明: TREE必须在BRAUN之前声明, ORDER要在PRIORITY_QUEUE之前声明。

由于我们的函子并不互相引用, 因此它们可以以任意顺序声明。现在可以立即声明函子PriorityQueue, 即使它的实现依赖于二叉树。这个函子是自包含的: 它以一个二叉树结构Tree作为形式参数, 并通过使用它来访问树的操作:

```
functor PriorityQueue (structure Item : ORDER
                      structure Tree : TREE)
  : PRIORITY_QUEUE =
  :
  :
  abstype t = PQ of Item.t Tree.tree
  :
  :
```

结构Flex (参见图4-3) 可以修改成为函子FlexArray, 其中将Braun作为它的形式参数。这个函子体类似于原来的结构声明, 但是树的操作现在变为子结构Braun.Tree的组件了。

```
functor FlexArray (Braun: BRAUN) : FLEXARRAY =
  :
  :
  val empty = Array(Braun.Tree.Lf, 0);
  :
  :
```

结构Braun也可以类似地修改为函子BraunFunctor, 其中将Tree作为它的形式参数。

```
functor BraunFunctor (Tree: TREE) : BRAUN = ...
```

⊖ 也可以直接描述类型tree, 见7.15节。

甚至结构`Tree`也可以改为函子：它具有空参数。

296

```
functor TreeFunctor () : TREE = struct ... end;
```

现在，所有的函子都已声明。

将函子链接在一起。在最后阶段，当所有代码都书写完毕后，就轮到应用函子了。每个结构都是通过将—个函子应用到已创建的结构上而建立的。—开始，将`TreeFunctor`应用到一个空参数表上来生成结构`Tree`。

```
structure Tree = TreeFunctor ();
> structure Tree : TREE
```

函子应用创建了结构`Braun`和`Flex`：

```
structure Braun = BraunFunctor (Tree);
structure Flex = FlexArray (Braun);
```

如前面那样声明结构`StringOrder`：

```
structure StringOrder = ...;
```

现在，像前面—样，可以通过函子应用声明结构`StringPQueue`：

```
structure StringPQueue =
  PriorityQueue (structure Item = StringOrder
                 structure Tree = Tree);
```

图7-4描绘了完整的系统，其中结构在弧形框中，函子在长方形框中。大多数结构都是由函子创建的，只有`StringOrder`是直接书写的。

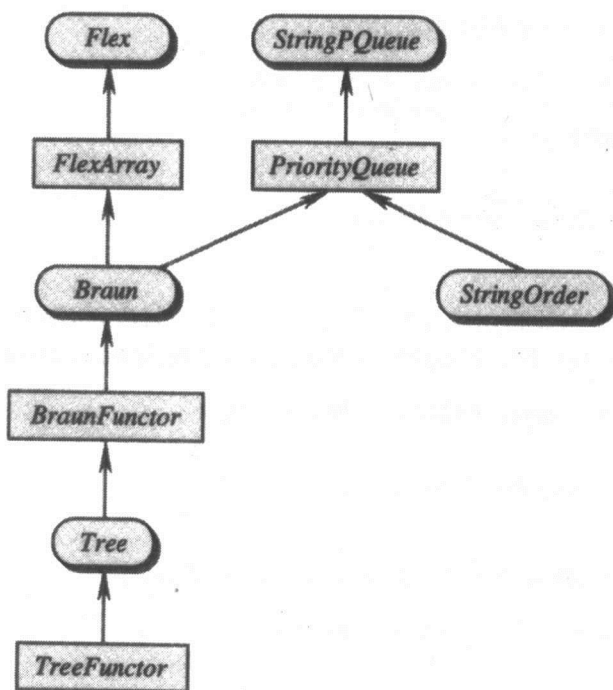


图7-4 涉及树的结构和函子

全函子方式的缺点应该是很明显的。所有那些函子声明堆满了整个代码，甚至给它们起名字都有点困难。共享约束成倍地增加。如果我们将这个例子继续下去的话，可以预见会需要很多的共享约束来保证这些结构共享相同的子结构`Tree`。对于某些ML系统，编译出来的代码可能是应有的两倍大，因为同样的代码既存在于函子中也存在于结构中。

比较好的折中方法是对主要的程序单元使用函子：也就是那些必须独立编写的部分。无论如何，这些部分大都是泛型的。下层的单元可以声明为结构。Biagioni等（1994）曾经用签名和函子将一个大型网络系统的各个层次组织起来，组件可以以多种方式连在一起来满足各种特殊的要求。

❶ 何时签名是自包含的？由一个描述引进的名字在签名的余下部分是可见的。签名`TREE`描述了类型`tree`，然后使用它以及类型`int`来描述`size`的类型。预定义的名字，像`int`和标准库结构，被称为普遍的（pervasive）：它们到处可见。一个名字如果出现在签名中，而又没有对其进行描述，那么这个名字就称为在签名中是自由的（free）。在`TREE`中唯一自由出现的名字是`int`。

David MacQueen为Standard ML模块（Harper等，1986）写了最初的计划书。他建议签名不能引用程序在其他地方声明的名字，除非这个名字代表另一个签名。签名可以引用在同一签名内描述的结构，但不能引用自由出现的结构。这样一来，所有的签名都是自包含的了，并且所有结构都包含了它所依赖的结构和类型。这个限制，称为签名闭合规则（signature closure rule），最终被放宽了，以给予程序员更大的自由。

在全函子方式中，结构是最后声明的。签名自然而然地遵守签名闭合规则，因为不存在它们可以引用的结构。在`BRAUN`中唯一自由的名字是普遍的类型`int`和签名`TREE`。如果

```
structure Tree : TREE
```

这一行被去掉的话，签名就要依赖某个已经声明了的结构`Tree`，因为签名在函数的类型里提到了`Tree.tree`。这是一种可以接受的程序设计方式，但它不是全函子方式，并且它也违反了签名闭合规则。

❶ 函子和签名约束。在全函子方式的程序设计中，每个函子只是通过形式参数来引用结构。除了参数结构的签名外，函子对其一无所知。就像给予实际结构一个不透明签名约束一样。

假设函子的形式参数包括了具有签名`DICTIONARY`的结构`Dict`。在函子体内，类型 `α Dict.t`看上去就像一个抽象类型：它没有可以用于模式匹配的构造子，并且相等测试也被禁止了。类型`Dict.key`也是类似的抽象，除非有一个共享约束将它等同于另外某个类型，否则我们将无法调用`Dict.lookup`。

练习 7.29 书写一个无参数函子，使其结果签名为`QUEUE`，要求函子实现队列的表示法3。

练习 7.30 为惰性表的抽象类型描述一个签名`SEQUENCE`，并通过书写结果签名为`SEQUENCE`的函子来实现这个类型。书写一个函子，以`QUEUE`和`SEQUENCE`的实例为参数，并声明像`depthFirst`和`breadthFirst`（5.17节）这样的函数。

练习7.31 列出在以下签名中自由出现的名字: *QUEUE1*、*QUEUE2*、*QUEUE3*、*DICTIONARY* (4.14节) 和 *FLEXARRAY* (4.15节)。

7.14 open声明

当结构嵌套的时候, 复合名字会有麻烦。在函子 *FlexArray* 体内, 二叉树类型被称为 *Braun.Tree.tree*, 它的构造子叫做 *Braun.Tree.Lf* 和 *Braun.Tree.Br*。类型及其构造子的作用没有变化, 但是使用这种构造子记法书写的模式很可能难以阅读。

尽管全函子方式令这个问题变得更糟了, 不过任何大程序中都会出现长的复合名字。幸好, 有许多方法来缩写这样的名字。

打开 (*open*) 结构将声明其中的项, 使得它们可以通过简单名字使用。*open* 声明的语法是

```
open Id
```

其中 *Id* 是结构的名字 (有可能是复合的)。一次只打开一层结构。在声明

```
open Braun;
```

之后, 我们可以书写 *Tree* 和 *Tree.Lf*, 而不是 *Braun.Tree* 和 *Braun.Tree.Lf*。如果我们继续声明

```
open Tree;
```

则可以书写 *Lf* 和 *Br*, 而不是 *Tree.Lf* 和 *Tree.Br*。在这个 *open* 声明的作用域内, *Lf* 和 *Br* 表示构造子, 它们不能被作为值来重新声明。

局部open声明。由于 *open* 是一个声明, *let* 或 *local* 构造可以用来限制它的作用域。回忆一下, *let* 使得声明私有于一个表达式, 而 *local* 则使得声明私有于另一个声明。

下面是局部 *open* 声明的一个例子, 它同时演示了 *open* 会被怎样地误用。函子 *FlexArray* 可能会如下使用 *local*:

```
functor FlexArray (Braun: BRAUN) : FLEXARRAY =
  struct
    local open Braun Braun.Tree
    in
      datatype 'a array = Array of 'a tree * int;
      val empty = Array(Lf, 0);
      fun length (Array(_, n)) = n;
      fun sub ...
      fun update ...
      fun delete ...
      fun loext ...
      fun lorem ...
    end
  end;
```

300 *open* 声明使 *Braun* 和 *Braun.Tree* 的组件可见, 而 *local* 声明则将它们的作用域限制在函子体内。我们不再需要书写复合名字了。

真不需要吗? 回想一下, 这个函子是基于 *Braun* 数组来实现弹性数组的。弹性数组的下标操作使用了 *Braun* 数组的下标操作。两个操作都叫做 *sub*, 为了避免名字冲突, 必须书写一个复合名字:

```

fun sub (Array(t,n), k) =
  if 0<=k andalso k<n then Braun.sub(t,k+1)
  else raise Subscript;

```

去掉上面的前缀*Braun*.会建立一个不合逻辑的*sub*递归调用, 以及一个类型错误。因此打开*Braun*并没有起到什么作用。另外, 也没有必要打开*Braun.Tree*, 因为函子体内只使用了两次这个前缀。

使用*let*的结构表达式。对于*open*的应用来说, *BraunFunctor*是较好的候选者, 其中大量使用了树的构造子 (见图7-5)。打开结构*Tree*使我们在类似*Br(w, Lf, Lf)*的表达式中不用再书写复合名字了。

```

functor BraunFunctor (Tree: TREE) : BRAUN =
  let open Tree in
    struct
      structure Tree = Tree;

      fun sub (Lf, _) = raise Subscript
        | sub (Br(v,t1,t2), k) =
          if k = 1 then v
          else if k mod 2 = 0
            then sub (t1, k div 2)
            else sub (t2, k div 2);

      fun update (Lf, k, w) =
          if k = 1 then Br (w, Lf, Lf)
          else raise Subscript
        | update (Br(v,t1,t2), k, w) =
          if k = 1 then Br (w, t1, t2)
          else if k mod 2 = 0
            then Br (v, update(t1, k div 2, w), t2)
            else Br (v, t1, update(t2, k div 2, w));

      fun delete (Lf, n) = raise Subscript
        | delete (Br(v,t1,t2), n) =
          if n = 1 then Lf
          else if n mod 2 = 0
            then Br (v, delete(t1, n div 2), t2)
            else Br (v, t1, delete(t2, n div 2));

      fun loext (Lf, w) = Br(w, Lf, Lf)
        | loext (Br(v,t1,t2), w) = Br(w, loext(t2,v), t1);

      fun lorem Lf = raise Size
        | lorem (Br(_,Lf,_)) = Lf
        | lorem (Br(_, t1 as Br(v,_,_), t2)) = Br(v, t2, lorem t1);

    end
  end;

```

图7-5 函子体内的*let open*例子

该函子使用了一种新的*let*构造, 这个构造是操作在结构上的。假设*Str*是需要用到声明*D*

的结构表达式, 那么对结构表达式

```
let D in Str end
```

进行求值将产生对 Str 求值的结果, 同时定义了 D 的作用域。如果 Str 具有形式`struct ... end`, 就像 $BraunFunctor$ 中的那样, 那么可以将其等价地写为 (如前面的例子)

```
struct
  local D in ... end
end
```

但是, 我们可以将`let`用在其他的结构表达式上, 例如函子应用。这一点在一个结构被多次使用时特别有用:

```
functor QuadOrder (O: ORDER) : ORDER =
  let structure OO = LexOrder (structure O1 = O
                                structure O2 = O)
  in LexOrder (structure O1 = OO
               structure O2 = OO)
end;
```

函子 $QuadOrder$ 以一个有序结构为参数, 并返回形如 $((w, x), (y, z))$ 的四元组字典序结构。它在内部建立了结构 OO , OO 定义了序偶的有序结构。

△ 结构中的中缀操作符。在结构中给出的中缀命令对外没有作用。当结构被打开时, 里面的名字作为普通标识符对外可见, 而不再被当作中缀操作符, 要恢复它的中缀状态需要新的中缀命令。复合名字永远不能成为中缀操作符, 只有简单名字允许出现在中缀命令中。

在所有结构之外给出的顶层中缀命令具有全局作用域。打开一个结构会绑定或重新绑定这些顶层操作符。

△ 使用`open`重新绑定标识符。打开多个结构可以一下子声明数以百计的名字。除非这些名字的含义详细明白, 否则我们可能想不起来它们属于哪个结构。使用`open`覆盖已有的绑定会特别混乱。

ML可能会提供 $Real32$ 、 $Real64$ 、 $Real96$ 等库结构, 它们实现了多种精度的标准浮点运算。这些结构都匹配签名 $REAL$, 里面描述了类型 $real$ 以及 $+$ 、 $-$ 、 \times 、 $/$ 等运算。

复合名字使得这些结构很难使用。下面64位版本的 $(a/x + x)/y$ 是很难看懂的:

```
Real64./ (Real64.+ (Real64./ (a,x), x), y)
```

使用一个局部的`open`声明可以恢复可读性:

```
let open Real64 in (a/x + x) / y end
```

不幸得很, 打开 $Real64$ 重新声明了所有的数值运算符, 消除了对它们的重载。我们不再能书写像 $n+1$ 这样的整数表达式了。整数运算仍旧可以通过它们所在的结构进行访问: 仍可书写 $Int.+(n, 1)$ 。但这是一种改进吗?

在顶层打开 $Real64$ 绝对是错误的。没有任何办法可以恢复重载。像上面那样在小

范围内打开`Real64`会比较好, 或者声明新的中缀运算符并将它们绑定到相应的64位函数上。

`open`之外的选择。就像这些例子所展示的那样, `open`可能导致麻烦。设计结构的目的之一是为了利用复合名字的长处。项的简单名字是太短了。像`Braun.sub`和`Flex.sub`这样的复合名字不单只是避免了冲突, 同时它们也提供了丰富的信息, 并强化了我们对于程序组织结构的了解。

通过声明缩写可以缩短复合名字, 而无须使用`open`。我们对于函子`FlexArray`的声明可以改进为:

```
functor FlexArray (Braun: BRAUN) : FLEXARRAY =
  struct
    local structure T = Braun.Tree
    in
      datatype 'a array = Array of 'a T.tree * int;
      val empty = Array(T.Lf, 0);
    end
    :
  end;
```

我们声明了结构`T`来对`Braun.Tree`进行缩写, 以此来代替打开这个结构的做法。虽然必须书写`T.tree`, 但这完全可以接受, 并且用`T.Lf`和`T.Br`所表示的模式也是简洁的。

有些程序员会认为复合标识符是不可忍受的, 至少对于那些大量使用的项来说如此。但是, 当只有其中几项被用到时犯不着去打开一个大型模块。这时, 一个`open`声明可以被几个单独的缩写替代:

```
type 'a queue      = 'a Queue.t;
val hd             = Queue.hd;
exception QEmpty   = Queue.E;
```

最后那行令`QEmpty`等同于`Queue.E`, 并且它还是一个构造子, 甚至可以出现在异常处理器中。在上面的异常绑定中, 其右侧必须是一个异常构造子的名字。

有选择地使用`open`。在4.13节中, 我们将数据类型`tree`声明在顶层, 以避免复合的构造子名字。该方式并不好, 所有类型和变量都应该归属于某一个结构。同样, 我们也不希望完全打开树结构。并且, 也没有类似异常绑定的方法来输出单个的数据类型构造子。

我们必须使用`open`, 但是可以有选择地进行。核心声明可以声明在一个子结构里, 在本例中, 它们是数据类型`tree`和函数`depth`。

```
structure Tree =
  struct
    structure Export =
      struct
        datatype 'a tree = Lf
          | Br of 'a * 'a tree * 'a tree;

        fun depth Lf      = 0
          | depth (Br(v,t1,t2)) = 1 + Int.max (depth t1, depth t2);
      end;
```

```

open Export;

fun size Lf          = 0
  | size (Br(v,t1,t2)) = 1 + size t1 + size t2;

:
end;

```

子结构`Export`包含了准备输出到顶层的项。它立即就被打开了，是为了将这些项输出到主结构中。以后我们可以输出这些核心项，同时保持其他组件只能通过复合名字访问：

```

open Tree.Export;
depth Lf;
> 0 : int
Tree.size Lf;
> 0 : int

```

这个思想的另一个变种就是在顶层声明核心项自己的结构。它可以称为`TreeCore`，并拥有自己的签名`TREECORE`。其他关于树的结构和签名可以引用这个核心。

练习 7.32 解释为什么`StrangePQueue`的声明是有效的。

```

functor StrangePQueue () =
  let structure UsedTwice = struct open StringOrder Tree end
  in   PriorityQueue (structure Item = UsedTwice
                      structure Tree = UsedTwice)
  end;

```

练习 7.33 下面的声明有什么效果？

```
open Queue3; open Queue2;
```

练习 7.34 下面对于多精度算术运算的尝试有什么错误？

```

functor MultiplePrecision (F: REAL) =
  struct
    fun half x = F./(x, 2.0)
  end;

```

7.15 签名和子结构

复杂的程序需要复杂的签名。当结构嵌套在一起的时候，它们的签名会因为过长的复合名字而变得混乱。假设我们为Braun数组的序偶声明一个签名，其中描述了一个匹配签名
BRAUN的子结构：

```

signature BRAUNPAIR0 =
  sig
    structure Braun: BRAUN
    val zip: 'a Braun.Tree.tree * 'b Braun.Tree.tree ->
              ('a*'b) Braun.Tree.tree
  :
  end;

```

复合名字写出来的`zip`类型变得不可读。和结构一样，有几种途径可以简化这样的签名。

避免子结构。严格来说，签名中完全不需要描述子结构，即便是它必须做到自包含。可以直接描述所有在`val`声明中出现的类型。从上面的签名中去掉结构`Braun`会令可读性更强：

```
signature BRAUNPAIR1 =
  sig
    type 'a tree
    val zip: 'a tree * 'b tree -> ('a*'b) tree
  :
end;
```

这个签名所描述的要比`BRAUNPAIR0`少很多。`Braun`的所有组件都不见了，并且`tree`被描述为一个纯粹的类型（`type`）。将`tree`描述为一个数据类型（`datatype`）则需要从签名`TREE`中复制它的完整描述，这种重复不是很好。

签名应该尽可能地小，因此`BRAUNPAIR1`可能是理想的。如果能做到其中所描述的组件可以独立于`Braun`中的组件使用，它也确实是理想的，换句话说，它应该在使用上是自包含的，而不仅仅是形式上没有自由的标识符。

签名中的共享约束。虽然签名应尽可能地小，但是它不应该过分小。如果每个`BRAUNPAIR1`的实例都需要伴随一个`BRAUN`的实例，那么标识它们的`tree`组件将需要一个共享约束。每个如下形式的函子头

```
functor PairFunctor0 (BP: BRAUNPAIR0)
```

的长度都会加倍：

```
functor PairFunctor1 (structure Braun: BRAUN
  structure BP: BRAUNPAIR1
  sharing type Braun.Tree.tree = BP.tree)
```

306

解决办法是在签名中同时描述子结构`Braun`和类型`tree`，并使用一个共享约束来将它们关联起来：

```
signature BRAUNPAIR2 =
  sig
    structure Braun: BRAUN
    type 'a tree
    sharing type tree = Braun.Tree.tree
    val zip: 'a tree * 'b tree -> ('a*'b) tree
  :
end;
```

与此签名匹配的结构必须声明类型`a tree`，以满足其中的共享约束：

```
type 'a tree = 'a Braun.Tree.tree
```

我们有了之前两种方案的优点。签名既可读又完备，也允许如`PairFunctor0`这样简单的函子头。

签名中的类型缩写。类型的共享约束对于目前的用途来说已经足够了，也就是说可以将签名中的复合名字缩短。但是，他们不能描述任意的类型缩写。共享约束是针对标识符的，而不是针对类型的，相应的共享描述是

```
tree = Braun.Tree.tree
```

而不是

```
'a tree = 'a Braun.Tree.tree
```

签名是可以使用类型缩写的。这是另一种缩短复合名字的办法:

```
signature BRAUNPAIR3 =
  sig
    structure Braun: BRAUN
    type 'a tree = 'a Braun.Tree.tree
    val zip: 'a tree * 'b tree -> ('a*'b) tree
  end;
```

要使一个结构匹配这样的签名, 必须在里面声明一个等价的类型缩写。

include描述。包含(include)一个签名意味着将其组件描述为直接属于当前签名的, 而不是属于一个子结构的。描述

```
include SIG
```

的效果相当于直接书写SIG的内容, 而不要首尾的sig...end框。我们的例子现在变为

```
signature BRAUNPAIR4 =
  sig
    include BRAUN
    val zip: 'a Tree.tree * 'b Tree.tree -> ('a*'b) Tree.tree
  end;
```

仍存在复合名字, 不过它们的长短是可以接受的, 这是因为子结构Braun不见了。子结构中的全部组件都已经结合在新的签名中了。这样一来, BRAUNPAIR4的实例同样匹配签名BRAUN。



将自己包入困境。多重包含是一种强有力的结构化技术。它可以和共享约束组合起来获得重命名的一些效果。比如说, 如果被包含的签名描述了类型to和from, 那么共享约束可以令它们等同于签名中的其他类型(Biagioni等, 1994)。标准库以类似的风格使用共享约束, 有时是为了将子结构中的组件重命名。

要避免包含有同样名字的签名: 这可能会赋予一个标识符重复或者矛盾的描述。过多地使用include会导致庞大平铺的签名, 掩盖了模块的层次结构。如果签名BRAUN自身使用了描述

```
include TREE
```

而不是

```
structure Tree : TREE
```

那么应该彻底不会有复合名字了。表面上看, 这会对改进可读性有所帮助, 但是三个不同结构的所有组件将被毫无组织地丢在一起。

模块参考指南

本节汇集了结构、签名和函子的概念, 并对模块语言整体作出了总结。本节从实际出发讲述整个语言, 包括一些较为偏僻的特性。首先重温一下基本定义。

结构是一些声明的集合, 典型组成内容是一些用作公共用途的项目。这里面可能包括类

型、值和其他结构。由于多个结构可以进一步组织成更大的结构，因此一个软件系统可以设计成为有层次的实体。一个结构可以被看成是一个单元，无论其内部如何复杂。

签名是由一些类型检测信息组成的，它们是结构中所声明的每一项的类型信息。其中列出了类型、值及其类型、结构及其签名。共享约束使不同子结构中的公共组件成为一体。不同的结构可以具有相同的签名，这种情况就像不同的值可以具有相同的类型一样。

函子是结构到结构的映射。函子体根据形式参数定义了一个结构，形式参数是由一个签名描述的。应用函子可以将一个实际结构替换进函子体内。函子使得程序单元可以分别编写，并且可以表达泛型单元。

模块则是结构或函子两者之一。

7.16 签名和结构的语法

本书的目的是教授程序设计技术，而不是完整地讲述Standard ML。然而模块涉及到了大量的语法，这里便系统地讲述它们的主要特性。

在语法定义中，可选语句被方括号所包括。可重复出现的语句（至少出现一次）则非严格地以三点的省略号（...）表示。例如，在

$$\text{exception } Id_1 \left[\text{of } T_1 \right] \text{ and } \dots \text{ and } Id_n \left[\text{of } T_n \right]$$

中，像“of T_1 ”这样的语句是可选的。关键字and分隔了联立声明。

签名的语法。一个签名形如

$$\text{sig } Spec \text{ end}$$

其中，Spec是类型、值、异常、结构和共享约束的描述。

形如

$$\text{val } Id_1:T_1 \text{ and } \dots \text{ and } Id_n:T_n$$

的值描述了命名为 Id_1, \dots, Id_n 的值以及它们相应的类型 T_1, \dots, T_n 。多个值和它们的类型可以联立声明。

类型可以如下（联立地）描述

$$\text{type } \left[TypeVars_1 \right] Id_1 \left[= T_1 \right] \text{ and } \dots \text{ and } \left[TypeVars_n \right] Id_n \left[= T_n \right]$$

如果 T_i 出现，其中 $i = 1, \dots, n$ ，那么 Id_i 则被描述为类型缩写。

允许相等测试的类型可以描述为

$$\text{eqtype } \left[TypeVars_1 \right] Id_1 \text{ and } \dots \text{ and } \left[TypeVars_n \right] Id_n$$

在type和eqtype描述中，类型都可以通过可选的类型变量（TypeVars）后跟一个标识符给出，和在类型声明的左边可以出现的完全相同。datatype（数据类型）描述和datatype声明是相同的。

异常，以及可选的相应类型，可以如下描述

$$\text{exception } Id_1 \left[\text{of } T_1 \right] \text{ and } \dots \text{ and } Id_n \left[\text{of } T_n \right]$$

结构以及它们的签名是如下描述的

```
structure  $Id_1:Sig_1$  and ... and  $Id_n:Sig_n$ 
```

共享约束则形如

```
sharing [type]  $Id_1 = Id_2 = \dots = Id_n$ 
```

标识符 Id_1, \dots, Id_n 被描述为共享的。如果出现关键字type则它们必须是类型标识符, 否则它们必须是结构标识符。共享约束可以出现在任何签名中, 不过它们大多出现在一个函子的形式参数表中, 这里形式参数是作为一个签名描述给出的。共享两个结构意味着共享它们中对应名字的类型。

包含描述形如

```
include  $SigId_1 \dots SigId_n$ 
```

每个 $SigId$ 都是一个签名标识符, 它们描述了该签名中的组件。

❶ where type限制词。最近提出了一种新的签名形式, 它允许在签名 Sig 中将类型标识符 Id_i 限制到已存在的类型 T_i 上:

```
 $Sig$  where type [ $TypeVars_1$ ]  $Id_1 = T_1$  and [ $TypeVars_n$ ]  $Id_n = T_n$ 
```

这种构造可以用于以非常精细的方式组合的签名场合。与不透明签名约束一起, 它提供了另一种声明抽象类型的方法。考虑下面的函子头:

```
functor Dictionary (Key: ORDER)
  >: DICTIONARY where type key = Key.t
```

这个函子的结果签名是一个DICTIONARY的抽象视图, 不过key被约束为参数结构Key所描述的键值类型。这纠正了7.5节末提到的关于不透明约束的“非此即彼”的局限。该函子体不再需要使用abstype了。

310

结构的语法。结构可以通过由struct和end括住的声明(其中可以声明子结构)来创建:

```
struct D end
```

结构也可以由函子的应用给出:

```
FunctorId (Str)
```

由FunctorId命名的函子被应用到结构Str上。这是函子应用的基本语法, 在我们开始的一些例子里被用到, 它只允许一个参数。传递多个参数需要一般形式的函子应用, 其中实际参数是一个声明:

```
FunctorId (D)
```

这是下面函子应用的缩写

```
FunctorId (struct D end)
```

并且, 这也类似于书写元组来作为函数参数以达到多个参数的效果。

结构中的局部声明形如

```
let  $D$  in  $Str$  end
```

对其求值将执行声明 D 并产生结构表达式 Str 的值。 D 的作用域局限于 Str 中。

结构可以有透明或不透明的签名约束：

```
 $Str$  :  $Sig$ 
```

```
 $Str$  :>  $Sig$ 
```

7.17 模块声明的语法

签名、结构和函子的声明都不允许出现在表达式中。结构可以声明在其他结构中，但是函子声明不能嵌套。

签名约束以:> Sig 形式给出，另外: Sig 也是可以的。

通过签名声明，可以用标识符 Id_1, \dots, Id_n 来表示相应的签名 Sig_1, \dots, Sig_n ：

```
signature  $Id_1 = Sig_1$  and ... and  $Id_n = Sig_n$ 
```

通过结构声明，就可以用标识符 Id_i 来表示结构 Str_i （并可选地指定了签名 Sig_i ）了，其中 $1 \leq i \leq n$ ：

```
structure  $Id_1 [ :> Sig_1 ] = Str_1$  and ... and  $Id_n [ :> Sig_n ] = Str_n$ 
```

函子声明的基本语法是

```
functor  $Id$  ( $Id' : Sig'$ ) [ :>  $Sig$  ] =  $Str$ 
```

其中 Id 是函子名， Id' 和 Sig' 是形式参数的名字和签名， Str 是函子体， Sig 是可选的签名约束。

函子声明的一般语法具有如下形式，它可以实现多个参数的效果

```
functor  $Id$  ( $Spec$ ) [ :>  $Sig$  ] =  $Str$ 
```

形式参数表由描述 $Spec$ 给出的。函子仍旧只有一个参数，它是一个结构，其签名由 $Spec$ 决定。这个形式参数在函子体内被隐式地打开，以使它的组件可见。

要点小结

- 结构不隐藏其内部表示。
- `abstype`（抽象类型）声明可以和结构及签名组合起来隐藏抽象数据类型的内部细节。
- 函子是以其他结构为参数的结构。
- 函子可以表达泛型算法，并允许程序单元自由组合。
- 共享约束可能是必要的，因为它能保证系统中的某些子组件是相同的。
- 复合名字可以通过多种方法来缩短，其中包括谨慎地使用`open`声明。

第8章 ML中的命令式程序设计

函数式程序设计有它自身的优点，但是我们这里要讲的是命令式程序设计，对于输入输出来讲，它是最自然的。有些程序特别关心对于状态的维护：比如说象棋程序必须跟踪棋子的位置！一些经典的数据结构，像散列表，是通过更新数组和指针的方式工作的。

Standard ML的命令式特性包括引用、数组和输入输出命令。虽然具有独特的ML风格，这些特性还是提供了对命令式程序设计普遍的支持。循环是通过递归或使用while构造来表达的。引用的方式则与C或Pascal中的指针不尽相同，首先它们是安全的。

命令式特性和函数式程序设计是兼容的。引用和数组可以用于具有纯函数式形式的函数和数据结构中。我们将使用引用来存储每个元素，以此编写序列（惰性表）。这样可以避免重复计算造成的浪费，这种重复计算是5.12节所述序列的一个缺陷。我们将在可变更数组的辅助下编写函数式数组（其中每次更新都将建立一个新的数组）。这种函数式数组的表示要比4.15节的二叉树方法高效得多。

典型的ML程序大部分是函数式的。它保留了许多函数式程序设计的优点，包括可读性，甚至是效率：垃圾收集对于不变的对象可能更快。甚至对于命令式程序来说，和传统的语言相比，ML也有它的优点。

本章提要

本章将讲述引用类型和数组，并举例说明它们在数据结构中的使用。对ML的输入输出技巧也会有所介绍。

本章包括以下几节：

- 引用类型。引用代表了存储位置，它可以被创建、更新和查看。不能创建多态引用，但是多态函数内可以使用引用。
- 数据结构中的引用。这里介绍了三个较大型的例子。我们修改了序列类型，使其在内部存储计算出来的元素。环形缓冲区说明了引用是如何表示链接数据结构的。V-数组在函数式数据结构中使用了命令式的程序设计技术。
- 输入和输出。库函数可以将字符串和像*real*这样的基本类型互相转换。通道作为字符流的载体将ML程序和输入输出设备联系起来。我们的例子包括了日期扫描，向HTML格式的转换，以及代码美化（pretty printing）。

313

引用类型

ML中的引用基本上就是存储器地址。它们对应于C、Pascal和其他类似语言中的变量，并且在链接数据结构中作为指针使用。对于流程控制结构，ML提供了while-do循环命令，另外if-then-else和case表达式也适用于命令式程序设计。本节结束时还将对引用类型和多态性之间的交互关系作出讲解。

8.1 引用及其操作

在ML程序执行的过程中，所有计算出来的值都将在某段时间内停留在机器的存储器中。对于函数式程序员来说，存储器只是计算机中的一个设备而已，除非存储空间不足，否则从不需要考虑这个设备。而对于命令式程序设计来说，存储空间是可见的。ML的引用表示了存储空间中某个位置的地址。每个位置都包含一个值，这个值可以通过赋值被替代。引用本身也是一个值，如果 x 具有类型 τ ，那么 x 的引用则可以写作 $\text{ref } x$ 并且具有类型 $\tau \text{ ref}$ 。

构造子 ref 可以创建引用，将它用在值 v 上时，则会分配一个新地址并以 v 作为初值，然后返回指向该地址的引用。虽然 ref 是一个ML函数，但是它不是数学意义上的函数，因为 ref 在每次调用时都返回一个新的地址。

将函数 $!$ 应用到引用上时则返回其指向的内容。这个操作称为反引用（dereferencing）。显然 $!$ 也不是一个数学函数，它的返回值取决于存储器的内容。

赋值 $E_1 := E_2$ 首先对 E_1 求值， E_1 必须返回一个引用 p ，然后对 E_2 求值，赋值将 E_2 的值存储在地址 p 中。从语法角度看， $:=$ 是一个函数，且 $E_1 := E_2$ 是个表达式，尽管它的作用是更新存储器。如同大多数更新机器状态的函数一样，赋值函数返回类型 unit 的值 $()$ 。

314

下面是这些基本语句的一个简单例子：

```
val p = ref 5 and q = ref 2;
> val p = ref 5 : int ref
> val q = ref 2 : int ref
```

这里声明了引用 p 和 q ，它们指向的初始内容分别为5和2。

```
(!p, !q);
> (5, 2) : int * int
p := !p + !q;
> () : unit
(!p, !q);
> (7, 2) : int * int
```

赋值将 p 的内容改变为7。请注意“内容”这个词！赋值并不改变 p 的值，这是一个固定的存储器地址，赋值改变的是这个地址里的内容。我们可以将 p 和 q 当作Pascal里面的整型变量，不同的是需要显式地进行反引用。必须书写 $!p$ 来取得 p 的内容，而 p 本身代表一个地址。

数据结构中的引用。由于引用也是ML里面的值，它们可以是元组、表等数据结构的一部分。

```
val refs = [p, q, p];
> val refs = [ref 7, ref 2, ref 7] : int ref list
q := 1346;
> () : unit
refs;
> [ref 7, ref 1346, ref 7] : int ref list
```

refs 的第一个元素和第三个元素表示的地址和 p 相同，而第二个元素所表示的地址则和 q 相同。ML编译器将引用打印成 $\text{ref } c$ ，而不是表示地址的数，其中 c 是引用的内容。因此对 q 赋值会影响 refs 的打印结果。我们来给表头元素赋值：

```
hd refs := 1415;
> () : unit
refs;
```

```
> [ref 1415, ref 1346, ref 1415] : int ref list
(!p, !q);
> (1415, 1346) : int * int
```

因为`refs`的首元素是`p`，所以对`hd refs`赋值就相当于对`p`赋值。

对于引用的引用也是允许的：

```
val refp = ref p and refq = ref q;
> val refp = ref (ref 1415) : int ref ref
> val refq = ref (ref 1346) : int ref ref
```

315

下面的赋值以`refp`的内容(`p`)的内容(1415)来更新`refq`的内容(`q`)。这里`refp`和`refq`就像Pascal里面的指针变量一样。

```
!refq := !(!refp);
> () : unit
(!p, !q);
> (1415, 1415) : int * int
```

引用的相等。ML的相等测试对于所有的引用类型都有效。两个同类型的引用只有在它们表示同一地址时才被看作是相等的。下面的测试验证了`p`和`q`是不同的引用，而`refs`的首元素和`p`相等，和`q`不等：

```
p=q;
> false : bool
hd refs = p;
> true : bool
hd refs = q;
> false : bool
```

在Pascal中，如果两个指针变量碰巧含有相同地址，那么它们的值是相等的，赋值会使这两个指针相等。ML中引用相等的记法可能有些特别，因为如果`p`和`q`是不同的引用，那么没有办法能使它们相同（除了重新声明）。在命令式语言中，所有变量都是可以更新的，这时指针变量其实涉及了两层引用。普通的指针相等记法就像是比较`refp`和`refq`的内容，`refp`和`refq`都是引用的引用：

```
!refp = !refq;
> false : bool
refq := p;
> () : unit
!refp = !refq;
> true : bool
```

起初，`refp`和`refq`含有不同的值，也就是`p`和`q`。将值`p`赋给`refq`则使`refp`和`refq`拥有相同的内容，两个“指针变量”引用的都是`p`。

当两个引用相等时，就像`p`和`hd refs`那样，给其中一个赋值也影响到另一个的内容。这种情况被称为别名化（aliasing），它将造成严重的混淆。别名化会出现在过程式语言中，在过程调用时，一个全局变量和一个形式参数可能表示了同一地址。

❶ 循环数据结构。引用的循环链在很多场合都可以见到。假设我们声明`cp`来引用整数后继函数，并在`cFact`函数中进行反引用。

316

```

val cp = ref (fn k => k+1);
> val cp = ref fn : (int -> int) ref
fun cFact n = if n=0 then 1 else n * !cp(n-1);
> val cFact = fn : int -> int

```

每次 $cFact$ 的调用都会取得 cp 的当前内容。最初这是个后继函数，并且有 $cFact(8) = 8 \times 8 = 64$ ：

```

cFact 8;
> 64 : int

```

我们将 cp 更新为包含 $cFact$ 。现在 $cFact$ 通过 cp 引用了自身。它变成了一个计算阶乘的递归函数：

```

cp := cFact;
> () : unit
cFact 8;
> 40320 : int

```

通过更新引用来建立一个图有时也称为“打结”。很多函数式语言的解释器就是按照上述方式实现递归函数的，也就是在运行环境中建立一个图。

练习 8.1 判断表达式的值是真还是假：if $E_1 = E_2$ then $ref E_1 = ref E_2$ 。

练习 8.2 声明函数 $+=$ 使得 $+= Id E$ 对于整数 E 具有 $Id := !Id + E$ 的效果。

练习 8.3 p 和 q 如前声明，解释当下列表达式在顶层输入时ML的回应：

```
p := !p+1      2*!q
```

8.2 控制结构

ML不区分命令和表达式。命令就是在求值时更新状态的表达式。大多数命令都具有类型 $unit$ 并返回 $()$ 。作为命令式语言来看，ML仅提供基本的控制结构。

条件表达式

```
if  $E$  then  $E_1$  else  $E_2$ 
```

可以看作是条件命令。它对 E 求值，有可能更新状态，如果布尔结果是 $true$ 则对 E_1 求值，否则对 E_2 求值。它会返回 E_1 或 E_2 的值，尽管在命令式程序设计中，这个结果通常都是 $()$ 。

注意，这种行为是由ML的一般性表达式原则所致，ML只有一种 if 构造。

类似地， $case$ 表达式也可作为控制结构：

```
case  $E$  of  $P_1 => E_1$  |  $\dots$  |  $P_n => E_n$ 
```

首先对 E 进行求值，有可能更新状态。然后，模式匹配会像通常那样选择某个表达式 E_i 。这个表达式将被求值，同样有可能更新状态，并且返回其结果值。

在函数调用 $E_1 E_2$ 和 n 元组 (E_1, E_2, \dots, E_n) 时，将从左到右地对表达式进行求值。如果 E_1 改变了状态，它可能会影响 E_2 的后果。

一系列的命令也可以通过表达式

$(E_1; E_2; \dots; E_n)$

来执行。当对这个表达式求值时，会对表达式 E_1, E_2, \dots, E_n 从左到右分别求值，整个表达式的结果是 E_n 的值，其他表达式的值被丢弃了。由于分号在ML中还有其他用处，上面这种构造必须用括号括起来，除非用在`let`表达式的主体部分：

```
let D in E1; E2; ...; En end
```

对于迭代，ML有`while`命令：

```
while E1 do E2
```

如果对 E_1 的求值结果为`false`，那么`while`就结束了；如果 E_1 的结果为`true`，那么就对 E_2 进行求值，并且再次执行`while`。精确地讲，`while`命令满足递归方程

```
while E1 do E2  $\equiv$  if E1 then (E2; while E1 do E2)
                      else ()
```

最后的返回值是`()`，因此对 E_2 进行求值只是为了获得其对状态改变的效果。

简单的例子。ML可以模仿过程式程序设计语言。下面的过程，除了显式的反引用（!操作）以外，都可以用Pascal和C来书写。函数`impFact`使用局部引用`resultp`和`ip`来计算阶乘，并返回`resultp`的最后内容。观察一下如何使用`while`命令来将循环体执行 n 次：

318

```
fun impFact n =
  let val resultp = ref 1
      and ip      = ref 0
  in   while !ip < n do (ip      := !ip + 1;
                        resultp := !resultp * !ip);
      !resultp
  end;
> val impFact = fn : int -> int
```

`while`循环体包括两个赋值。在每次迭代中都将`ip`的内容加一，并利用`ip`的新内容来更新`resultp`的内容。

虽然调用`impFact`分配了新的引用。但是这个状态的变化对外是不可见的。`impFact(E)`的值是 E 值的数学函数。

```
impFact 6;
> 720 : int
```

在过程式语言中，过程可以有引用参数（变参），以便通过它们来修改调用者的变量。在Standard ML中，引用参数就是具有引用类型的形式参数。我们可以将`impFact`变换为过程`pFact`，其中以`resultp`作为引用参数。

```
fun pFact (n, resultp) =
  let val ip = ref 0
  in   resultp := 1;
      while !ip < n do (ip      := !ip + 1;
                        resultp := !resultp * !ip)
  end;
> val pFact = fn : int * int ref -> unit
```

调用`pFact(n, resultp)`将 n 的阶乘赋值给`resultp`：

```
pFact (5,p);
> () : unit
p;
> ref 120 : int ref
```

这两个函数演示了命令式风格，然而纯粹的递归函数是最清晰的，大概也是最快的计算阶乘的方法。更为现实的命令式程序将在本章稍后介绍。

库函数支持。标准库声明了一些用于命令式程序设计的顶层函数。函数`ignore`忽略它的参数值并返回`()`。下面是一个典型用例：

319

```
if !skip then ignore (TextIO.inputLine file)
  else skip := true;
```

上面的输入输出命令返回一个字符串，而赋值则返回`()`。调用`ignore`丢弃了字符串，防止类型`string`和`unit`发生冲突。对`ignore`的参数求值只是为了得到它的副作用，这里则是跳过文件的下一行。

有时，在执行某个命令之前必须取得表达式的值。例如，如果`x`包含0.5且`y`包含1.2，我们可以这样交换它们的内容：

```
y := #1 (!x, x := !y);
> () : unit
(!x, !y);
> (1.2, 0.5) : real * real
```

交换之所以成功，是因为序偶中参数的求值是依次进行的。函数`#1`返回第一个分量，^⑨也就是`x`原有的内容。中缀库函数`before`为这个技巧提供了更好的语法。它只是简单地返回它的第一个参数。

```
y := (!x before x := !y);
```

表算子`app`将命令应用到了表的每一个元素上。例如，下面的函数将同一个值赋给了引用列表中的每一个成员：

```
fun initialize rs x = app (fn r => r:=x) rs;
> val initialize = fn : 'a ref list -> 'a -> unit
initialize refs 1815;
> () : unit
refs;
> [ref 1815, ref 1815, ref 1815] : int ref list
```

显然，`app fl`类似于`ignore(map fl)`，不同的是它不用构造结果列表。顶层版本的`app`来自于结构`List`。其他库结构，包括`ListPair`和`Array`，也定义了相应版本的`app`。

异常和命令。当一个异常被抛出时，正常的执行流程被打断了。如同4.8节中讲述的那样，这时要选择一个异常处理器，并将控制转移到那里。这可能是危险的，异常可能会随时出现，产生一种不正常的状态。下面的异常处理器可以捕捉任何异常，整理当前状态，然后重新抛出该异常。变量`e`是一个能匹配所有异常的普通的模式（具有类型`exn`）：

320

```
handle e => ( ... (*整理动作*) ... ; raise e)
```

⑨ 2.9节对形如`#k`的选择函数进行了解释。

注意：大多数命令都可以返回类型`unit`的值`()`。从现在起，我们的会话中将省去令人厌烦的回应

```
> () : unit
```

练习 8.4 表达式 $(E_1; E_2; \dots; E_n)$ 和`while E_1 do E_2` 在ML中都属于派生形式，也就是说它们是通过翻译成其他表达式来定义的。请叙述（对于它们的）合适的翻译。

练习 8.5 书写命令式版本的`sqrroot`函数，通过牛顿-拉夫森方法来计算实数平方根（2.17节）。

练习 8.6 书写命令式版本的`fib`函数，通过高效的方法来计算斐波那契数（2.15节）。

练习 8.7 联立赋值

$$V_1, V_2, \dots, V_n := E_1, E_2, \dots, E_n$$

首先对各个表达式求值，然后将它们的值赋给相应的引用。例如，`x, y := !y, !x`交换了`x`和`y`的内容。书写一个ML函数来完成联立赋值。该函数应具有多态类型 $(\alpha \text{ ref}) \text{ list} \times \alpha \text{ list} \rightarrow \text{unit}$ 。

8.3 多态引用

自从引用被引入到程序设计语言中起，它就是一个臭名昭著的不安全因素。通常都不保留引用内容的类型信息，一个字符码有可能被解释为一个实数。Pascal避免了这种错误，保证每个引用只能包含一个固定类型的值，对于每一个类型 τ ，都采用一个不同的类型“指向 τ ”。然而，在ML中解决这个问题是比较难的：当 τ 是一个多态类型时， $\tau \text{ ref}$ 又意味着什么呢？除非我们特别小心，否则这种引用中的内容可能过一段时间就会发生变化。

一个虚构的会话。下面这个非法的会话演示了如果只是将引用粗糙地加入到类型系统中将会引起怎样的错误。我们从声明恒等函数开始：

```
fun I x = x;
> val I = fn : 'a -> 'a
```

由于`I`是多态的，它可以应用到任何类型的参数上。现在我们建立一个对`I`的引用：


```
val fp = ref I;
> val fp = ref fn : ('a -> 'a) ref
```

根据它的多态类型 $(\alpha \rightarrow \alpha) \text{ ref}$ ，我们应该能将`fp`的内容应用到任何类型的参数上：

```
(!fp true, !fp 5);
> (true, 5) : bool * int
```

同时，它的多态类型使我们可以将类型为`bool → bool`的函数赋值给`fp`：

```
fp := not;
!fp 5;
```



将`not`应用到整数5上面是一个运行时的类型错误，但是我们认为ML应该在编译时就能检测出所有的类型错误。显然是某个地方出了问题，但是哪儿出了问题呢？

多态性和替换。在不考虑命令式的情况下，重复地对于一个表达式求值总是得到相同的结果。声明`val Id = E`使得`Id`成为这个表达式结果的同义词。忽略效率因素，我们可以在所有`Id`

出现的地方用 E 来替换。

例如，这里有两个多态声明，一个是函数的声明，另一个是表的声明：

```
let val I = fn x => x in (I true, I 5) end;
> (true, 5) : bool * int
let val nil = [[]] in (["Exeter"]::nil, [1415]::nil) end;
> ([["Exeter"], []], [[1415], []])
> : string list list * int list list
```

替换掉声明既不影响返回值，也不影响类型：

```
((fn x => x) true, (fn x => x) 5);
> (true, 5) : bool * int
(["Exeter"]::[], [1415]::[]);
> ([["Exeter"], []], [[1415], []])
> : string list list * int list list
```

322

现在让我们来看看当那个虚构的会话被包装在`let`表达式中时，ML是怎样回应的：

```
let val fp = ref I
in ((!fp true, !fp 5), fp := not, !fp 5) end;
> Error: Type conflict: expected bool, found int
```

谢天谢地，ML不接受这个表达式，并且给出了含义确切的错误信息。如果将声明替换掉又会发生什么呢？

```
((!(ref I) true, !(ref I) 5), (ref I) := not, !(ref I) 5);
> ((true, 5), (), 5) : (bool * int) * unit * int
```

对表达式的求值没有出错。但是这个替换完全改变了表达式的含义。原来的表达式分配了一个引用 fp ，其初始内容为 I ，然后两次提取它的内容，接着更新并最后提取新的内容。修改后的表达式分配了四个不同的引用，每个都被赋给初始内容 I 。中间的赋值是没有意义的，它更新了一个其他地方没有用到的引用。

问题的关键是，重复地调用`ref`总是产生新的引用。我们声明`val fp = ref I`是期望每个 fp 出现的地方都能表示同一个引用：也就是同一个存储地址。替换并没有考虑到 fp 的共享。多态性在处理每个标识符的类型时都用它的定义表达式的类型来替换，因此要假设替换是有效的。

问题出在共享上，而不是副作用上。我们必须对多态引用的创建加以控制，而不是控制它们的赋值。

多态值的声明。语法值 (syntactic value) 是一种简单到不能创建引用的表达式，它具有以下几种形式：

- 文字常数，比如3，是语法值。
- 标识符也是一种，因为它代表了其他某个已经处理过的声明。
- 语法值可以经由其他语法值通过使用元组、记录 and 构造子 (`ref` 除外) 的形式构建。
- 采用`fn`记法的函数是语法值，即便它的函数体内使用了`ref`，因为函数体在函数被调用之前不会执行。

对于`ref`和其他函数的调用不是语法值。

如果 E 是一个语法值，那么多态声明`val Id = E`在替换下是等价的。这种声明的多态和往常一样：每个 Id 的出现都可以具有 E 的多态类型的不同实例。每个`fun`声明也都如此处理，因

323

为它只是带有fn记法的val声明的简短形式，而fn记法是语法值。

如果 E 不是个语法值，那么声明 $\text{val } Id = E$ 有可能创建引用。为了顾及共享，每个 Id 的出现必须具有同一类型。如果该声明出现在一个let表达式中，那么每个 E 类型中的类型变量在整个let表达式体中都被固定。表达式

```
let val fp = ref 1
in fp := not; !fp 5 end;
```

是不合法的，因为 $\text{ref } l$ 涉及类型变量 α ，它不能同时代表 bool 和 int 。同样，表达式

```
let val fp = ref 1
in (!fp true, !fp 5) end;
```

也是不合法的。但是这个表达式是安全的：如果我们可以对其求值，结果会是 $(\text{true}, 5)$ ，而没有运行错误。单态的版本是合法的：

```
let val fp = ref 1
in fp := not; !fp true end;
> false : bool
```

当 E 不是语法值时，顶层的多态声明是被禁止的，因为类型检测器不能预测之后 Id 将被如何使用：

```
val fp = ref 1;
> Error: Non-value in polymorphic declaration
```

单态的类型约束可以使顶层声明成为合法的。下面的表达式不再是创建多态引用：

```
val fp = ref (1: bool -> bool);
> val fp = ref fn` : (bool -> bool) ref
```

命令式的表翻转。现在来看一个实际的多态性的例子。函数 irev 以命令式的方式翻转了一个表。它用一个引用对表进行扫描，同时用另一个引用反向积累表的元素。

```
fun irev l =
  let val resultp = ref []
      and lp      = ref l
  in while not (null (!lp)) do
      (resultp := hd(!lp) :: !resultp;
       lp      := tl(!lp));
    !resultp
  end;
> val irev = fn : 'a list -> 'a list
```

变量 lp 和 $resultp$ 都具有类型 $(\alpha \text{ list}) \text{ ref}$ ，类型变量 α 在let表达式体内是固定的。ML接受 irev 作为一个多态函数，因为它通过fun声明的。

我们也可以验证 irev 的确是多态的：

```
irev [25,10,1415];
> [1415, 10, 25] : int list
irev (explode("Montjoy"));
> [#"y", #o, #"j", #"t", #"n", #o, #"M"]
> : char list
```

它可以像标准函数 rev 一样使用。

多态异常。虽然异常并不涉及存储器，但是它们也需要某种形式的共享。看看下面的错误程序：

```
exception Poly of 'a;          (* 不合法!! *)
(raise Poly true) handle Poly x => x+1;
```

如果这个表达式可以被求值的话，它会试图对`true + 1`进行求值，这是一个运行错误。当声明一个多态异常时，ML要保证它只被作为一种类型使用，就像限制值声明一样。顶层的异常必须是单态的，并且局部异常中的类型变量要被固定。

值多态性的局限。像在5.4节提到过的那样，将多态限制在语法值上的做法排除了一些很自然的多态声明。大多数情况下，这些都可以很容易地被修正，比如说使用`fun`替代`val`：

```
val length = foldl (fn (_,n) => n+1) 0;    (* 被拒绝 *)
fun length l = foldl (fn (_,n) => n+1) 0 l;  (* 被接受 *)
```

编译时类型检测必须有保留地假设运行时可能发生的事情。类型检测拒绝了很多本来可以安全执行的程序。尽管表达式`hd [5, true] + 3`的类型是错误的，但是可以很安全地对它求值而得到8。大多数现代语言都使用编译时类型检测，而程序员也接受这些限制，以避免运行时的类型错误。

❶ 多态引用的历史。很多人都曾经对多态引用进行过研究，但是通常都承认是Mads Tofte解决了这个问题。早期的ML编译器禁止了所有的多态引用：函数`ref`只能具有单态类型。Tofte最初的建议在ML定义中被采用，它比现在用到的要宽松。特殊的“弱”类型变量用于跟踪命令式特性，只有这些变量在`val`声明中被加以限制。Standard ML of New Jersey则使用了一个实验性质的方案，其中弱类型变量具有用数值标记的弱度。

弱类型变量产生了复杂且不直观的类型。它们对`irev`和`rev`赋予了不同的类型，这妨碍了两个函数的互换使用。最糟的是，它们使得在实现相应结构之前，很难事先写出签名。

Wright (1995) 建议平等地对待所有的`val`声明——其效果是使所有的类型变量变弱。纯函数式代码会被当做命令式代码看待。程序员会容忍这种限制吗？Wright使用了一个修改过的类型检测器来测试过大量他人书写的ML代码，结果经过他的限制之后只产生了很少的错误，并且那些错误是易于修补的。因此，经过充分考虑后，将这个建议推荐给了ML。

在Standard ML中对于多态引用的类型检测大概是安全的。Tofte (1990) 证明了它对于一个ML子集的正确性，并且没有理由去怀疑它对于整个语言的正确性。Greiner (1996) 研究过一个简化版的New Jersey系统。Harper (1994) 讲述了对于这种证明的一个简单方案。Standard ML经过了非常谨慎的定义，以避免不安全和其他语义上的缺陷，从这方面看，这个语言实际上是独具一格的。

练习 8.8 这个表达式合法吗？`WI`是做什么的？

```
let fun WI x = !(ref x)
in (WI false, WI "Clarence") end
```

练习 8.9 下面哪些表达式是合法的？如果可以求值的话，哪个会导致运行时类型错误？

```
val funs = [hd];
val l    = rev [];
val l'   = tl [3];
val lp   = let fun nilp x = ref [] in nilp() end;
```

数据结构中的引用

任何算法方面的书都会讲述像表和树这样的递归数据结构。这些数据结构和ML的递归数据类型（见第4章）有所不同，主要表现在：前者的递归用到了显式的链接域，或者说指针。这些指针可以被更新，使其能重新链接已有的数据结构，并创建圈。

引用类型和递归数据类型结合在一起可以实现这样的链接数据结构。本节给出两个这样的例子：双循环链接表和一个高效的函数式数组。我们从一个简单一点的引用使用开始：不是作为链接域，而是用来存储先前的计算结果。

326

8.4 序列，或惰性表

在5.12节给出的表示法下，序列的尾部是个计算另一序列的函数。每当查看尾部时，一个代价可能颇高的函数调用就被重复一次。我们可以不采用这种低效的做法。通过引用来表示序列尾部，这个引用开始包含了一个函数，后来被更新为函数的结果。这样实现的序列利用了可变更的存储空间，但是从外面看，它仍是纯函数式的。

序列的抽象类型。结构`ImpSeq`实现了惰性表，见图8-1。类型 αt 有三个构造子：`Nil`用于构造空序列，`Cons`用于构造非空序列，而`Delayed`则允许对尾部进行延时求值。一个具有如下形式的序列

$$\text{Cons}(x, \text{ref}(\text{Delayed } xf))$$

是从 x 开始，并以序列 $xf()$ 作为它的其余元素的，其中 xf 具有类型 $\text{unit} \rightarrow \alpha t$ 。注意`Delayed xf`包含在一个引用单元中。应用`force`将它更新为包含 $xf()$ 的返回值，去掉`Delayed`。这样做会有一些开销，但是一旦序列元素再次被访问，则效率会显著提高。

函数`null`测试序列是否为空，而`hd`和`tl`返回序列的首元素和尾元素。由于`tl`调用了`force`，因此序列最外层的构造子不可能是`Delayed`。在结构`ImpSeq`内部，序列上的函数可以利用模式匹配；而在外部，则必须使用`null`、`hd`和`tl`，这是因为构造子都被隐藏了。不透明的签名约束保证了结构可以产生一个抽象类型：

```
signature IMP_SEQUENCE =
sig
  type 'a t
  exception Empty
  val empty    : 'a t
  val cons     : 'a * (unit -> 'a t) -> 'a t
  val null     : 'a t -> bool
  val hd       : 'a t -> 'a
  val tl       : 'a t -> 'a t
  val take     : 'a t * int -> 'a list
  val toList   : 'a t -> 'a list
  val fromList : 'a list -> 'a t
```

```

val @      : 'a t * 'a t -> 'a t
val interleave : 'a t * 'a t -> 'a t
val concat  : 'a t t -> 'a t
val map     : ('a -> 'b) -> 'a t -> 'b t
val filter  : ('a -> bool) -> 'a t -> 'a t
val cycle   : ((unit -> 'a t) -> 'a t) -> 'a t
end;

```

```

structure ImpSeq :> IMP_SEQUENCE =
struct
  datatype 'a t = Nil
                | Cons  .of 'a * ('a t) ref
                | Delayed of unit -> 'a t;

  exception Empty;

  fun delay xf = ref (Delayed xf);

  val empty = Nil;

  fun cons(x,xf) = Cons(x, delay xf);

  fun force xp =
    case !xp of
      Delayed f => let val s = f()
                    in xp := s; s end
    | s => s;

  fun null Nil      = true
    | null (Cons _) = false;

  fun hd Nil      = raise Empty
    | hd (Cons(x,_)) = x;

  fun tl Nil      = raise Empty
    | tl (Cons(_,xp)) = force xp;

  fun take (xq, 0)      = []
    | take (Nil, n)     = []
    | take (Cons(x,xp), n) = x :: take (force xp, n-1);

  fun          Nil @ yq = yq
    | (Cons(x,xp)) @ yq =
        Cons(x, delay(fn()=> (force xp) @ yq));

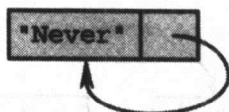
  fun map f Nil      = Nil
    | map f (Cons(x,xp)) =
        Cons(f x, delay(fn()=> map f (force xp)));

  fun cycle seqfn =
    let val knot = ref Nil
    in knot := seqfn (fn()=> !knot); !knot end;
end;

```

图8-1 使用引用的惰性表

循环序列。函数cycle通过打结建立了一个循环序列。下面是一个尾部是其自身的序列：



这个序列的表现就像是无穷序列"Never", "Never", ..., 但是只占用了计算机中很少量的空间。它是通过如下调用建立的:

```
ImpSeq.cycle(fn xf => ImpSeq.cons("Never", xf));
> - : string ImpSeq.t
ImpSeq.take(it, 5);
> ["Never", "Never", "Never", "Never", "Never"]
> : string list
```

当应用cycle于某个seqfn函数时, 它创建了引用knot, 并将它(包装成一个函数)提供给seqfn。seqfn的结果是一个序列, 这个序列随着其中元素的计算最终将引用knot中的内容。更新knot使它包含这个序列本身, 这样便建立了一个圈。

循环序列能够以奇妙的方式计算斐波那契数。定义add函数, 它将两个整数序列逐项相加, 返回和序列。为了说明引用的多态性, add要基于另一个函数来编写, 该函数将两个序列合并为一个序偶序列:

```
fun pairs(xq,yq) =
  ImpSeq.cons((ImpSeq.hd xq, ImpSeq.hd yq),
    fn()=>pairs(ImpSeq.tl xq, ImpSeq.tl yq));
> val pairs = fn
> : 'a ImpSeq.t * 'b ImpSeq.t -> ('a * 'b) ImpSeq.t
fun add (xq,yq) = ImpSeq.map Int.+ (pairs(xq,yq));
> val add = fn
> : int ImpSeq.t * int ImpSeq.t -> int ImpSeq.t
```

斐波那契数序列可以利用cycle来定义:

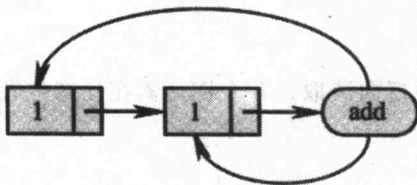
```
val fib = ImpSeq.cycle(fn fibf =>
  ImpSeq.cons(1, fn()=>
    ImpSeq.cons(1, fn()=>
      add(fibf(), ImpSeq.tl(fibf())))));
> val fib = - : int ImpSeq.t
```

327
329

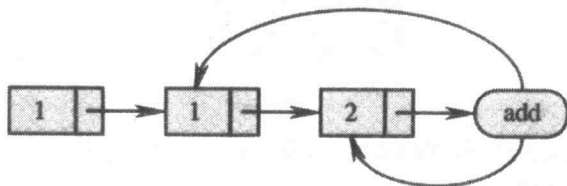
这个定义是循环的。序列由1, 1开始, 其余的元素通过将这个序列和其尾部序列相加获得:

`add (fib, ImpSeq.tl fib)`

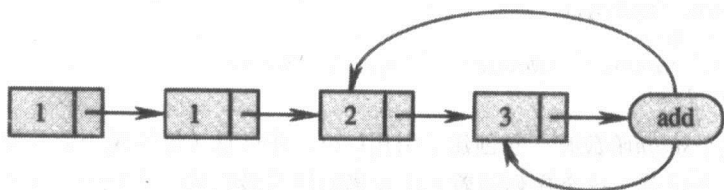
开始, fib可以画成这样:



当通过tl (tl fib)访问fib的第三个元素时, add调用计算出2, 然后force将序列更新成下面的样子:

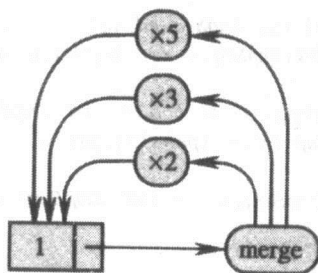


当下一个元素被访问时, *fib* 变成



由于序列是循环的, 并且保留计算出的元素, 因此每个斐波那契数只计算一次。这是相当快的。如果斐波那契数是用5.12节的序列来递归定义的话, 计算第 n 项的代价将是以 n 为指数的。

练习 8.10 海明 (Hamming) 问题是按升序枚举所有形如 $2^i 3^j 5^k$ 的整数。声明一个由这些数组成的循环序列。提示: 声明一个合并升序序列的函数, 并参考下图:



练习 8.11 实现函数 *iterates*, 在给定 f 和 x 时, 创建序列 $[x, f(x), f(f(x)), \dots, f^k(x), \dots]$ 的一个循环表示。

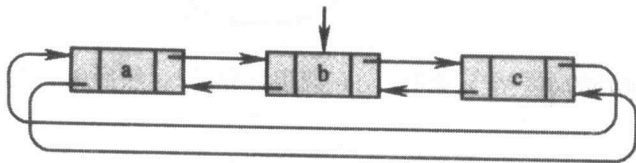
练习 8.12 讨论证明循环序列是否正确的难度, 正确是指可以产生满足给定描述的值序列。评论下面这个序列:

```
val fib2 = ImpSeq.cycle (fn fibf =>
  ImpSeq.cons(1, fn () => add(fibf(), ImpSeq.tl(fibf()))));
```

练习 8.13 编写在结构 *ImpSeq* 中缺少的, 而又在其签名中描述了的函数, 也就是 *toList*、*fromList*、*interleave*、*concat* 和 *filter*。

8.5 环形缓冲区

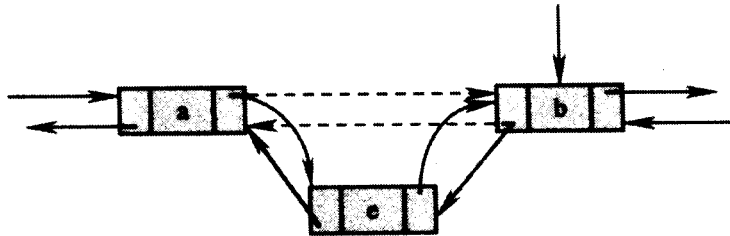
双向链接表可以从正向或反向读取, 并允许元素在任意位置插入和删除。当它闭合回到自身时就是循环的:



这种可变更数据结构有时称为环形缓冲区 (ring buffer)，应该为大多数程序员所熟悉。在此，我们实现它以对Standard ML中的引用和过程式语言中的指针变量作一比较。我们来定义一个抽象类型，它具有如下签名：

```
signature RINGBUF =
  sig
    eqtype 'a t
    exception Empty
    val empty    : unit -> 'a t
    val null     : 'a t -> bool
    val label    : 'a t -> 'a
    val moveLeft : 'a t -> unit
    val moveRight: 'a t -> unit
    val insert   : 'a t * 'a -> unit
    val delete   : 'a t -> 'a
  end;
```

环形缓冲区具有类型 αt ，是一个指向双向链接表的引用。新的环形缓冲区可以通过调用函数`empty`来创建。函数`null`测试环形缓冲区是否为空，`label`则返回当前结点的标签，此外，`moveLeft`和`moveRight`分别将指针移向当前结点的左边和右边。如下所示，`insert(buf, e)`将标签为`e`的结点插入到当前结点的左边。两个链接项被重定向到新的结点，它们原先的指向以虚线箭头表示，它们最终的指向以阴影箭头表示：



函数`delete`移走了当前结点，并将指针向右移动。函数的返回值是被删除结点的标签。

图8-2所示代码和用Pascal书写出来的差不多。双向链接表的每个结点都具有类型 αbuf ，其中含有一个标签和指向其左右结点的引用，给定一个结点，函数`left`和`right`就可以分别返回这两个引用。

构造子`Nil`表示空表，同时也起到了占位的作用，就像Pascal的`nil`指针一样。单有类型 αbuf 的构造子`Node`是无法创建任何 αbuf 值的。看一下`insert`的代码，当创建第一个结点时，它的左右指针开始都只包含`Nil`，然后它们被更新为包含结点自身的表。

请牢记ML中的引用相等和通常概念下的指针相等不同。函数`delete`必须检测是否正在删除缓冲区中仅有的一个结点。要确定`Node(lp, x, rp)`是否是仅有的结点不能通过检测`lp = rp`是否成立来完成，这和Pascal程序员所期望的不同。在正确构造出来的缓冲区中，这个检测总是不成立的，每个链接域必定是不同的引用，这样才可能对它们分别进行更新。进行`left(!lp) = lp`的测试是正确的。如果左侧的结点（也就是`!lp`）和当前结点具有相同的左链接，那么它们就是同一个结点，因此也就是缓冲区中仅剩的结点。

下面是一个环形缓冲区的小型演示。首先，我们来创建一个空缓冲区。由于对`empty`的调用不是一个语法值，我们必须将它的结果约束到某个单态类型上，在这里是`string`。（和空序

列`ImpSeq.empty`比较, 后者不包含引用, 并且是多态的。)

```
val buf: string RingBuf.t = RingBuf.empty();
> val buf = - : string RingBuf.t
RingBuf.insert(buf, "They");
```

```
structure RingBuf :> RINGBUF =
  struct
    datatype 'a buf = Nil | Node of 'a buf ref * 'a * 'a buf ref;
    datatype 'a t   = Ptr of 'a buf ref;
    exception Empty;

    fun left (Node(lp,_,_)) = lp
      | left Nil           = raise Empty;

    fun right (Node(_,_ ,rp)) = rp
      | right Nil            = raise Empty;

    fun empty() = Ptr(ref Nil);

    fun null (Ptr p) = case !p of
      Nil          => true
    | Node(_,x,_) => false;

    fun label (Ptr p) = case !p of
      Nil          => raise Empty
    | Node(_,x,_) => x;

    fun moveLeft (Ptr p) = (p := !(left(!p)));
    fun moveRight (Ptr p) = (p := !(right(!p)));

    fun insert (Ptr p, x) =
      case !p of
        Nil          =>
          let val lp = ref Nil
            and rp = ref Nil
            val new = Node(lp,x,rp)
            in lp := new; rp := new; p := new end
      | Node(lp,_,_) =>
          let val new = Node(ref(!lp), x, ref(!p))
            in right(!lp) := new; lp := new end;

    fun delete (Ptr p) =
      case !p of
        Nil          => raise Empty
      | Node(lp,x,rp) =>
          (if left(!lp) = lp then p := Nil
           else (right(!lp) := !rp; left(!rp) := !lp; p := !rp);
           x)

    end;
```

图8-2 作为双向链接表的环形缓冲区

如果只进行`insert`和`delete`, 那么环形缓冲区就像是一个可变更的队列, 元素在插入后可以按同样的顺序取出。

```

RingBuf.insert(buf, "shall");
RingBuf.delete buf;
> "They" : string
RingBuf.insert(buf, "be");
RingBuf.insert(buf, "famed");
RingBuf.delete buf;
> "shall" : string
RingBuf.delete buf;
> "be" : string
RingBuf.delete buf;
> "famed" : string

```

练习 8.14 修改`delete`，返回一个布尔值而不是标签：`true`表示删除后的缓冲区为空，而`false`则相反。

练习 8.15 下列哪些相等测试可以用于检测`Node(lp, x, rp)`是否为环形缓冲区中仅有的结点？

$!lp = !rp \quad \text{right}(!lp) = lp \quad \text{right}(!lp) = rp$

练习 8.16 比较下面的插入函数和`insert`，下面的函数是否有长处或短处？

```

fun insert2 (Ptr p, x) =
  case !p of
    Nil          => p := Node(p, x, p)
  | Node(lp, _, _) =>
      let val new = Node(lp, x, p)
      in right(!lp) := new; lp := new end;

```

练习 8.17 编写另一版本的`insert`来将新结点插入到当前结点的右侧而不是左侧。

练习 8.18 证明如果可以声明类型为 $\alpha \text{ RingBuf.t}$ （强类型变量）的值，那么将会产生运行错误。

练习 8.19 类型 $\alpha \text{ RingBuf.t}$ 上的相等测试有什么用处？

333
334

8.6 可变更的数组和函数式的数组

*Standard ML*定义中并没有提及数组，但是标准库中提供了结构`Array`，它具有如下签名：

```

signature ARRAY =
  sig
    eqtype 'a array
    val array      : int * 'a -> 'a array
    val fromList   : 'a list -> 'a array
    val tabulate   : int * (int -> 'a) -> 'a array
    val sub        : 'a array * int -> 'a
    val update     : 'a array * int * 'a -> unit
    val length     : 'a array -> int
    ...
  end;

```

每个数组都有固定的大小。一个 n 元素的数组允许下标范围为0到 $n-1$ 。数组操作在超出边界时抛出异常`Subscript`，在试图创建负大小或特别大的数组时抛出异常`Size`。[⊖]

⊖ 这些异常在库结构`General`中声明。

下面简单讲述了主要的数组操作:

- `array(n, x)` 创建一个 n 元素数组, 并将 x 存入每个单元。
- `fromList [x0, x1, ..., xn-1]` 创建一个 n 元素数组, 并将 x_k 存入单元 k 中, 其中 $k = 0, \dots, n-1$ 。
- `tabulate(n, f)` 创建一个 n 元素数组, 并将 $f(k)$ 存入单元 k 中, 其中 $k = 0, \dots, n-1$ 。
- `sub(A, k)` 返回数组 A 中单元 k 的内容。
- `update(A, k, x)` 将数组 A 中单元 k 的内容更新为 x 。
- `length(A)` 返回数组 A 的大小。

数组是可变更的对象, 它的行为很像引用。数组间总是允许相等测试: 两个数组相等当且仅当它们是同一对象时。也可以创建数组的数组, 就像Pascal一样, 用来作为多维数组。

❶ 标准库的聚合结构。类型为 α array 的数组可以被更新。不变数组提供对静态数据的随机访问, 并且可使函数式程序效率更高。库结构 `Vector` 声明了不变数组的类型 α vector (向量)。该结构提供了与 `Array` 几乎相同的操作, `update` 除外。函数 `tabulate` 和 `fromList` 是创建向量, 而 `Array.extract` 则是从数组中提取向量。

由于类型 α array 和 α vector 都是多态的, 对每个元素, 它们需要额外一层间接。单态数组和向量可以更为紧凑地表示。库签名 `MONO_ARRAY` 描述了另一个类型 `elem` 上的可变更数组类型 `array`。签名 `MONO_VECTOR` 也是类似的, 描述了不变数组类型 `vector`。多个库函数结构满足这个签名, 它们给出了字符数组和浮点数数组等。

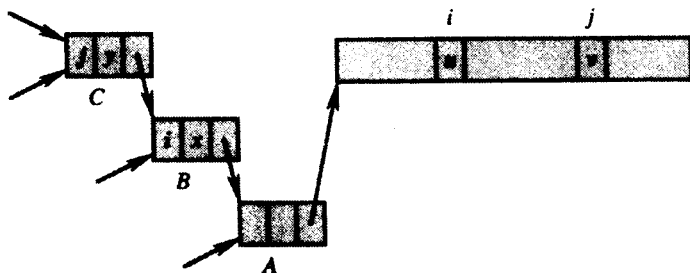
库认为数组、向量甚至是表都出于同一概念: 聚合。相应的操作尽可能地保持一致。数组和表一样具有 `app` 和折叠算子。函数 `Array.fromList` 将一个表转换成数组, 相对应的逆操作也很容易编写:

```
fun toList l = Array.foldr op:: [] l;
```

表和数组一样有 `tabulate` 函数。两者都支持下标操作, 从零开始索引, 并且一旦上界溢出都抛出异常 `Subscript`。

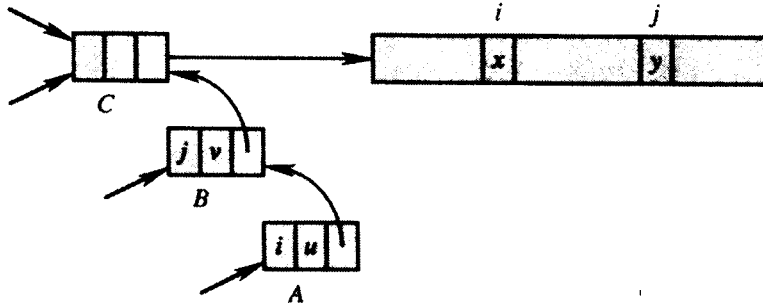
表示函数式数组。Holmström和Hughes开发了一种混合式的函数式数组表示法, 用到了可变更数组和关联表。由 $(index, contents)$ (索引和内容) 序偶组成的关联表具有函数式的更新操作: 简单地在表前增加一个新序偶而已。更新是很快, 但是查找则需要费时的搜索。引入一个称作向量 (vector) 的可变更数组使查找相对快些 (Aasa等, 1988)。

开始时, 函数式数组由一个向量表示。更新操作在这个向量前面建立了一个关联表, 用以指出向量的当前内容和各个数组值之间的区别。考虑函数式数组 A 的两个单元 i 和 j , $i \neq j$, 并且假定 $A[i] = u$ 以及 $A[j] = v$ 。现在来进行一些函数式的更新。将 x 存放于位置 i 从而由 A 得到 B , 将 y 存放于位置 j 从而由 B 得到 C :



指向A、B和C的其他链接也显示出来了，这些是来自进一步更新所创建的数组。这些数组形成了一棵树，称作版本树（version tree），因为它的结点是向量的不同“版本”。和通常的树不同，它的链接项是指向根结点的，而不是从根结点指出去的。树的根结点是A，这是一个链接到向量上的哑结点。哑结点只含有指向向量的直接链接，它的作用是简化根重整操作。

版本树的根重整。虽然C具有正确的值，也就是 $C[i] = x$ 及 $C[j] = y$ ，并且其他值和A一样，但是对C的查找可能要比一般情况慢。如果C是最常使用的向量版本，那么版本树的根结点应该移到C。从C出发到向量的那些链接项要翻转过来，并且那些结点所指出的更新要在向量中执行，而先前向量单元的内容则被记录在那些结点中。



这个操作并不影响函数式数组的各个值，但是查找A变慢了，而查找C变快了。哑结点现在是C。版本树中引用A、B或C的那些结点经过了查找时间的变化，但不是值的变化。根重整并不需要定位其他那些结点。如果没有其他对A或B的引用，那么ML的存储分配器将回收它们的空间。

一种实现。图8-3所示的是一个关于版本树数组的ML结构声明，简称v-数组。它匹配下面的签名：

```
signature VARRAY =
  sig
    type 'a t
    val array      : int * 'a -> 'a t
    val reroot     : 'a t -> 'a t
    val sub        : 'a t * int -> 'a
    val justUpdate : 'a t * int * 'a -> 'a t
    val update     : 'a t * int * 'a -> 'a t
  end;
```

不透明的签名约束隐藏了v-数组的表示方法，包括相等测试。内部的相等测试只是比较存储对象的一致性，这并不是函数式的特性。

v-数组的类型是 αt ，它有构造子Modif和Main。Modif结点（用于修改）是具有四个域的记录。存放v-数组的上限是为了进行下标检测。其他域是一些引用，分别指向一个索引、一个元素和下一个v-数组，这些域在根重整时会被更新。Main结点里包含了可变更数组。

调用`array(n, x)`构造出一个v-数组，它由一个向量和一个哑结点组成。递归函数`reroot`完成根重整。下标操作`sub(va, i)`在节点中搜索下标 i ，如果需要则在向量中查找该下标。函数`justUpdate`只是建立一个新结点，而`update`则在这个操作之后紧接着对新数组进行了根重整。库异常`Subscript`和`Size`可能被显式地抛出，也可能从`Array`操作中抛出。

```

structure Varray :> VARRAY =
  struct
    datatype 'a t = Modif of {limit : int,
                              index : int ref,
                              elem : 'a ref,
                              next : 'a t ref}
      | Main of 'a Array.array;

    fun array (n,x) =
      if n < 0 then raise Size
      else Modif{limit=n, index=ref 0, elem=ref x,
                  next=ref (Main (Array.array (n,x)))};

    fun reroot (va as Modif{index, elem, next, ...}) =
      case !next of
        Main _ => va (* 已经到达根 *)
      | Modif _ =>
          let val Modif{index=bindex, elem=belem, next=bnext, ...} =
              reroot (!next)
              val Main ary = !bnext
          in bindex := !index;
             belem := Array.sub(ary, !index);
             Array.update(ary, !index, !elem);
             next := !bnext;
             bnext := va;
          va
        end;

    fun sub (Modif{index,elem,next,...}, i) =
      case !next of
        Main ary => Array.sub(ary,i)
      | Modif _ => if !index = i then !elem
                    else sub(!next,i);

    fun justUpdate(va as Modif{limit,...}, i, x) =
      if 0<=i andalso i<limit
      then Modif{limit=limit, index= ref i, elem=ref x, next=ref va}
      else raise Subscript;

    fun update(va,i,x) = reroot(justUpdate(va,i,x));
  end;

```

图8-3 作为版本树的函数式数组

程序通常都会以单线索的方式使用v-数组，在每次更新后丢弃数组先前的值。在这种情况下，我们应该在每次更新后进行根重整。如果一个函数式数组的很多个版本同时存在，那么版本树可能会变得低效，因为只有一个版本能用向量表示。这时，我们应当将函数式数组表示为二叉树，就像在4.15节中那样。另外，二叉树还可以允许数组增长或缩短。

i v-数组的实验结果。上面的代码是基于Aasa等（1988）。对于多个单线索算法来说，他们发现v-数组比其他函数式数组的表示法要更高效。最好的情况，v-数组可以在常数时间内完成查找和更新，尽管比起可变更数组还是要慢些。基于v-数组的快速排序并不比基于表的快速排序来得快，这暗示了数组应该保留给需要随机访问的任务使用。对于元素的顺序处理来说，表的效率更高。

练习 8.20 回想3.7节中的函数`allChange`。在数组的辅助下，书写一个可以高效确定下列表达式值的函数

```
length(allChange([], [5,2], 16000));
```

练习 8.21 向结构`Varray`中添加函数`fromList`，实现根据非空表的内容创建`v`-数组。

练习 8.22 向结构`Varray`中添加函数`copy`，使得`copy(va)`创建一个值和`va`相同的新`v`-数组。

练习 8.23 为二维可变更数组声明一个结构`Array2`，其中包含的组件和`Array`的类似。

练习 8.24 为二维`v`-数组声明一个结构`Varray2`，其中包含的组件和`Varray`的类似。

练习 8.25 哑结点中的内容是什么？其他的`v`-数组表示法能否省去这个结点？

337
}

339

输入和输出

输入输出可能是一个令人厌烦的问题。读取数据和打印结果相对于中间的计算来说是琐碎的，特别是这些操作必须服从于常见操作系统中的那些武断的功能。输入输出将安全、类型化的、基本是函数式的计算环境与面向字节的、命令式的设备相接。小部分人对ML定义中仅描述了一个很吝啬的相关原语集感到惊讶。Algol 60的定义根本没有提到输入输出。

ML库补充了这方面的不足，描述了若干输入输出模型和数十个操作。它也提供了字符串处理函数来对输入进行扫描，对输出进行格式化。我们有选择地研究其中一部分。

像树这样的链接数据结构的输入输出造成了特殊的困难。将它们摊平成字符串通常会破坏对于子树的广泛共享，导致指数爆炸。像Poly/ML中的那种永久存储可以高效地存放任意的数据。这种设施很难找到，而且也缺乏弹性。

8.7 字符串处理

库提供了广泛的函数用于处理字符串和子串。结构`Int`、`Real`和`Bool`（还有其他的）包含了在基本值和字符串之间进行转换的函数。主要的函数有`toString`、`fromString`、`fmt`和`scan`。对于这些函数，库在每个适当的结构中都声明了专用版本，而不是在顶层重载它们。你也可以在自己的某些结构中声明这些函数。

转换成字符串。函数`toString`按默认格式将它的参数表示成一个字符串：

```
Int.toString (~23 mod 10);
> "7" : string
Real.toString Math.pi;
> "3.14159265359" : string
Bool.toString (Math.pi = 22.0/7.0);
> "false" : string
```

340

结构`StringCvt`支持更为精细的格式化。你可以指定显示多少个小数位，以及将结果字符串补足到需要的长度。你还可以选择基数。例如，DEC PDP-8使用八进制记法，并将整数补足到四个数字：

```
Int.fmt StringCvt.OCT 31;
> "37" : string
StringCvt.padLeft #"0" 4 it;
> "0037" : string
```

像`String.concat`这样的操作可以把格式化的结果与其他文本组合在一起。

将字符串进行转换。函数`fromString`将字符串转换到基本的值。这个函数允许比较随意地输入，数值的表示比ML程序中合法的那些要宽松得多，例如，在`~`以外，也接受`+`和`-`：

```
Real.fromString "+.6626e-33";
> SOME 6.626E~34 : real option
```

对字符串进行由左到右的扫描，并且忽略后面多余的字符。可能检测不到用户的错误：

```
Int.toString "1o24";
> SOME 1 : int option
Bool.fromString "falsetto";
> SOME false : bool option
```

无论输入可以多么随意，总有些字符串是没有意义的：

```
Int.fromString "My master's mind";
> NONE : int option
```

分解字符串。既然`fromString`忽略剩余的字符，我们又如何将字符串中的一系列值转换出来？库结构`String`和`Substring`为扫描提供了有用的函数。函数`String.tokens`从字符串中提取出一个词法单元列表。词法单元是一些非空的子串，它们由一个或多个分隔符隔开。分隔符由一个类型为`char → bool`的谓词定义，结构`Char`中包含有识别字母（`isAlpha`）、空格（`isSpace`）和标点符号（`isPunct`）的谓词。这里有一些调用例子：

```
String.tokens Char.isSpace
  "What is thy name? I know thy quality.";
> ["What", "is", "thy", "name?",
  > "I", "know", "thy", "quality."] : string list

String.tokens Char.isPunct
  "What is thy name? I know thy quality.";
> ["What is thy name", " I know thy quality"]
> : string list
```

这样，我们就可以将输入的字符串分解成它的组成部分，然后把它们传给`fromString`。函数`dateFromString`（见图8-4）用于将形如`dd-MMM-yyyy`的日期解码。它以三个横线分隔的词法单元作为输入，利用`Int.fromString`来分析日和年，用`String.Substring`将月缩短为三个字母。如果月份找不到，或者出现异常，那么函数返回`NONE`，异常`Bind`可能在三个地方出现。

```
dateFromString "25-OCTOBRE-1415-shall-live-forever";
> SOME (25, "OCT", 1415) : (int * string * int) option
dateFromString "2L-DECK-18o5";
> SOME (2, "DEC", 18) : (int * string * int) option
```

我们看到`dateFromString`就像其他的`fromString`一样允许随意地输入。

从字符源中扫描。在几个库结构中都可以发现扫描函数`scan`，它们提供对文本处理的精确控制。这些函数接受任意以函数方式给出的字符源，而不只是一个字符串。如果可以书写函数

$$getc : \sigma \rightarrow (\text{char} \times \sigma) \text{ option}$$

那么类型 σ 就可以作为一个字符源。调用`getc`或者返回`NONE`，再或者返回下一个字符和之后

的字符源所组成的包。

```
val months = ["JAN", "FEB", "MAR", "APR", "MAY", "JUN",
              "JUL", "AUG", "SEP", "OCT", "NOV", "DEC"];

fun dateFromString s =
  let val sday::smon::syear::_ = String.tokens (fn c => c = #" -") s
      val SOME day = Int.fromString sday
      val mon      = String.substring (smon, 0, 3)
      val SOME year = Int.fromString syear
  in if List.exists (fn m => m=mon) months
    then SOME (day, mon, year)
    else NONE
  end
  handle Subscript => NONE
       | Bind      => NONE;
```

图8-4 从字符串中扫描日期

这些`scan`函数读入一个基本值，尽可能地取走字符，并将其余内容留给后续处理。例如，我们将表定义为一个字符源：

```
fun listGetc (x::l) = SOME (x,l)
  | listGetc []    = NONE;
> val listGetc = fn : 'a list -> ('a * 'a list) option
```

`scan`函数都是柯里的，以字符源作为它们的第一个参数。整数的`scan`函数之前还需要给定一个基数，`DEC`表示十进制。我们来扫描一些带有错误的输入：

```
Bool.scan listGetc (explode "mendacious");
> NONE : (bool * char list) option
Bool.scan listGetc (explode "falsetto");
> SOME (false, [#"t", #"t", #"o"]): (bool * char list) option
Real.scan listGetc (explode "6.626x-34");
> SOME (6.626, [#"x", #" -", #"3", #"4"])
> : (real * char list) option
Int.scan StringCvt.DEC listGetc (explode "1o24");
> SOME (1, [#"o", #"2", #"4"]): (int * char list) option
```

打错了的字符`x`和`o`并没有阻止（它们前面的）数的扫描，但是这些字符被留在输入中。这种错误是可以被检测出来的，方法是查看输入是否读完，或者接下来的字符是不是一个期望的分隔符。在后一种情况下，可以跳过分隔符，接着扫描后面的值。

`fromString`函数易于使用，但是会漏掉一些错误。`scan`函数则构成了坚固的输入处理基础。

练习 8.26 声明函数`writeCheque`，用于打印支票上的钱数。调用`writeCheque w (dols, cents)`应能将元和分的和表达为正好具有宽度`w`的域。例如，`writeCheque 9 (57, 8)`应该返回字符串"`$***57.08`"。

练习 8.27 书写函数`toUpper`，用于将字符串中的所有字母转成大写，其他字符保持不变。

341
343

(库结构`String`和`Char`中有相关的函数。)

练习 8.28 重写上面的例子，利用子串代替字符列表作为字符源。(库结构`Substring`里声明了有用的函数，包括`getc`。)

练习 8.29 利用`scan`函数来编写扫描日期的函数。它应能接受一个任意的字符源。(库结构`StringCvt`里有相关的函数。)

8.8 文本输入输出

ML最简单的输入输出模型支持文本文件上的命令式操作。流(stream)连接外部文件(或设备)和程序以传输字符。将输入流连接到数据的生产者上，例如键盘，可以从中读取字符直到该生产者结束这个流。将输出流连接到数据的消费者上，例如打印机，可以向它送出字符直到程序结束这个流。

输入和输出操作都属于结构`TextIO`，它的签名实际上是下面的一个扩展：

```
signature TEXTIO =
  sig
    type instream and outstream
    exception Io of {name: string, function: string, cause: exn}
    val stdIn      : instream
    val stdOut     : outstream
    val openIn     : string -> instream
    val openOut    : string -> outstream
    val closeIn    : instream -> unit
    val closeOut   : outstream -> unit
    val inputN     : instream * int -> string
    val inputLine  : instream -> string
    val inputAll   : instream -> string
    val lookahead  : instream -> char option
    val endOfStream : instream -> bool
    val output     : outstream * string -> unit
    val flushOut   : outstream -> unit
    val print      : string -> unit
  end;
```

下面是这些项的简单描述。详见库文档。

- 输入流具有类型`instream`，而输出流具有类型`outstream`。这两个类型都不允许相等测试。
- 异常`Io`指示了某个低级的操作失败。它带有受影响的文件名、原始的函数名以及被抛出的原始异常。
- `stdIn`和`stdOut`是标准输入输出流，在交互式会话中，它们被连接到终端。
- `openIn(s)`和`openOut(s)`创建连接到文件的流，文件名由`s`给出。
- `closeIn(is)`和`closeOut(os)`终止一个流，断开它与对应文件的联系。之后这个流就不能再传输字符了。输入流可能会被它对应的设备终止，例如在遇到文件结束时。
- `inputN(is, n)`从流`is`中取走最多`n`个字符，并将它们作为字符串返回。如果在流关闭之前所取出的字符不足`n`个，那么就只返回这些字符。
- `inputLine(is)`从流`is`中读取下一行文本，并将它作为一个以换行结束的字符串返回。如果流`is`已经关闭，那么返回空串。

344

- `inputAll(is)` 读取流 `is` 的全部内容，并将它们作为字符串返回。通常，它是读入整个文件，并不适用于交互式的输入。
- `lookahead(is)` 返回下一个字符，如果存在的话，它并不从流 `is` 中取走该字符。
- `endOfStream(is)` 当流 `is` 在结束符之前没有更多的字符时返回 `true`（真）。
- `output(os, s)` 向流 `os` 中写入字符串 `s` 中的字符，前提是流尚未被关闭。
- `flushOut(os)` 向最终的目的文件送出仍等待于系统缓冲区中的字符。
- `print(s)` 将字符串 `s` 中的字符写到终端上，也可以通过 `output` 和 `flushOut` 来实现。函数 `print` 在顶层可用。

上面的输入操作可能会阻塞：等待将延续到需要的字符出现或是流被关闭。

假设文件 `Harry` 中存放了亨利五世（Henry V）的一些话，摘自他在阿让库尔战役（battle of Agincourt）之前不久对法国人说的话：

```
My people are with sickness much enfeebled,
my numbers lessened, and those few I have
almost no better than so many French ...
But, God before, we say we will come on!
```

（译文：我手下的人，有好一些害了病，力量大大削弱了，数目也减少了，而留下来的为数不多的人，又几乎并不比那许多法国人高出一筹……可是老天在上，我们说了，我们是非来不可的。——莎士比亚《亨利五世》）

将 `infile` 作为 `Harry` 的输入流。我们看一眼第一个字符：

```
val infile = TextIO.openIn("Harry");
> val infile = ? : TextIO.instream
TextIO.lookahead infile;
> SOME #"M" : char option
```

345

调用 `lookahead` 并不会在文件中前进。但是现在我们要提取出十个字符作为一个字符串，然后读取该行剩下的字符。

```
TextIO.inputN (infile, 10);
> "My people " : string
TextIO.inputLine infile;
> "are with sickness much enfeebled;\n" : string
```

调用 `inputAll` 读入文件剩余的部分作为一个长且难懂的字符串，然后将它输出到终端：

```
TextIO.inputAll infile;
> "my numbers lessened, and those few I have\nalmo# : string
print it;
> my numbers lessened, and those few I have
> almost no better than so many French ...
> But, God before, we say we will come on!
```

最后会发现文件显示出我们正处于文件结尾，因此要把文件关闭：

```
TextIO.lookahead infile;
> NONE : char option
TextIO.inputLine infile;
> "" : string
TextIO.closeIn ;
```

当对流的访问结束后关闭它们可以保存系统资源。

8.9 文本处理的例子

几个小例子将演示怎样在ML中处理文本文件。真正进行输入输出的代码量很小，有点令人感到意外，而类似`String.tokens`的字符串处理函数却做了大部分的工作。

批处理输入输出。我们的第一个例程是读入一系列行并打印每个单词首字母。单词是以空格分隔的语法单元，通过下标操作可以得到它们的首字符，并用`implode`将首字符连接成一个字符串：

```
fun firstChar s = String.sub(s,0);
> val firstChar = fn : string -> char
val initials = implode o (map firstChar) o
    (String.tokens Char.isSpace);
> val initials = fn : string -> string
initials "My ransom is this frail and worthless trunk";
> "Mritfawt" : string
```

346

函数`batchInitials`在给定输入和输出流时，不断地逐行读取输入，并且将每行的单词首字符写入到输出流中，直到输入流被读完。

```
fun batchInitials (is, os) =
  while not (TextIO.endOfStream is)
  do TextIO.output(os, initials (TextIO.inputLine is) ^ "\n");
> val batchInitials = fn
> : TextIO.instream * TextIO.outstream -> unit
```

将`infile`作为全新的Harry输入流，对其应用`batchInitials`：

```
val infile = TextIO.openIn("Harry");
> val infile = ? : TextIO.instream
batchInitials(infile, TextIO.stdOut);
> Mpawsme
> mnlafIh
> anbtSMF.
> BGbwswwco
```

输出显示在终端上，这是因为`stdOut`被指定为输出流。

交互式输入输出。只要将`stdIn`作为第一个参数传入即可使`batchInitials`从终端读取字符。但是程序的交互式版本应该在暂停下来准备接受输入时显示一个提示符。一个朴素的尝试就是在调用`inputLine`之前调用`output`（输出提示符）：

```
while not (TextIO.endOfStream is)
do (TextIO.output(os, "Input line? ");
    TextIO.output(os, initials(TextIO.inputLine is) ^ "\n"));
```

但是这样会将打印提示符延至读取输入之后。这里有两个错误：(1) 必须调用`flushOut`来保证输出确实能被显示出来，而不是存放在缓冲区中。(2) 必须在调用`endOfStream`之前打印提示符，因为那个函数可能会阻塞，因此必须将提示符代码移至`while`和`do`关键字之间。下面是个改良版本：

```

fun promptInitials (is, os) =
  while (TextIO.output(os, "Input line? ");
        TextIO.flushOut os;
        not (TextIO.endOfStream is))
  do TextIO.output(os, "Initials: " ^
                    initials(TextIO.inputLine is) ^ "\n");
> val promptInitials = fn
> : TextIO.instream * TextIO.outstream -> unit

```

347

回忆一下，对表达式(E_1 ; E_2 ; ...; E_n)求值就是分别依次对 E_1 , E_2 , ..., E_n 求值，并返回 E_n 的值。我们可以在测试循环条件之前执行任何命令。在下面的例子执行中，提供给标准输入（用户键入）的文本被标上下划线：

```

promptInitials(TextIO.stdIn, TextIO.stdOut);
> Input line? If we may pass, we will;
> Initials:   Iwmpww
> Input line? If we be hindered ...
> Initials:   Iwbh.
> Input line?

```

最后的输入是Control-D，它终止了输入流。这并不妨碍我们以后继续从`stdIn`中读取字符。类似地，在遇到一个文件的结尾后，某个其他进程可能会扩展那个文件。调用`endOfStream`可能会在此时返回`true`（真），而稍后返回`false`（假）。

如果输出流总是终端，那么使用`print`可以进一步简化`while`循环：

```

while (print "Input line? "; not (TextIO.endOfStream is))
do print ("Initials: " ^ initials(TextIO.inputLine is) ^ "\n");

```

转换到HTML。下一个例子只进行简单的输入输出，重点说明子串的使用。类型`substring`的值由一个字符串`s`和两个整数`i`和`n`表示，它代表了`s`中从位置`i`开始的`n`个字符的段。子串高效地支持特定形式的文本处理，其中只有很少的复制和边界检测。一个子串可以被分解成词法单元，或者将其从左到右进行扫描，操作的结果本身也是子串。

我们的任务是将剧本由纯文本转换成HTML，它是在互联网上使用的超文本标记语言。图8-5显示的是一个典型的输入。段落是以空行分隔的。每句对白自成一段；相应的输出必须插入一个`<P>`标记。每段的第一行给出了人物的名称，后跟一个句号；相应的输出中必须把名称用标记``和``括住以突出强调。为了保留断行，转换必须在每段的诸行后加上`
`标记。

```

Westmoreland. Of fighting men they have full three score thousand.

Exeter. There's five to one; besides, they all are fresh.

Westmoreland. O that we now had here
But one ten thousand of those men in England
That do no work to-day!

King Henry V. What's he that wishes so?
My cousin Westmoreland? No, my fair cousin:
If we are marked to die, we are enough
To do our country loss; and if to live,
The fewer men, the greater share of honour.

```

图8-5 转换之前的无格式输入

函数`firstLine`处理每段的第一行，将名称和行的余下部分分离。它使用了库结构`Substring`中的三个组件，分别是`all`、`splitl`和`string`。调用`all s`建立一个代表整个字符串`s`的子串。调用`splitl`从左到右扫描字符串，在`name`中返回第一个句号之前的子串，在`rest`中返回原始串的剩余部分。调用`string`将这些子串转换成字符串，以便可以将它们连接到其他含有标记的字符串上。

```
fun firstLine s =
  let val (name, rest) =
        Substring.splitl (fn c => c <> #".") (Substring.all s)
    in "\n<P><EM>" ^ Substring.string name ^
        "</EM>" ^ Substring.string rest
    end;
  > val firstLine = fn : string -> string
```

在下面的例子中观察标记的位置：

```
firstLine "King Henry V. What's he that wishes so?";
> "\n<P><EM>King Henry V</EM>. What's he that wishes so?"
> : string
```

函数`htmlCvt`接受一个文件名参数并打开输入输出流。它的主循环是递归函数`cvt`，该函数每次转换一行，并监视是否遇到段落的第一行。空字符串表示输入结束，而空行（只含有一个换行符）则另起一段。其他的行根据是否为第一行作出相应的转换。程序输出转换好的行，并继续处理过程。

```
fun htmlCvt fileName =
  let val is = TextIO.openIn fileName
        and os = TextIO.openOut (fileName ^ ".html")
        fun cvt _ "" = ()
          | cvt _ "\n" = cvt true (TextIO.inputLine is)
          | cvt first s =
              (TextIO.output (os,
                              if first then firstLine s
                              else "<BR>" ^ s);
               cvt false (TextIO.inputLine is));
        in cvt true "\n"; TextIO.closeIn is; TextIO.closeOut os
        end;
  > val htmlCvt = fn : string -> unit
```

最后，`htmlCvt`关闭输入输出流。关闭输出流将导致在缓冲区存放的文本确实实的写到文件中。图8-6给出了在浏览器中显示的转换好的文本。

Westmoreland. Of fighting men they have full three score thousand.

Exeter. There's five to one; besides, they all are fresh.

Westmoreland. O that we had here
But one ten thousand of those men in England
That do no work to-day!

King Henry V. What's he that wishes so?
My cousin Westmoreland? No, my fair cousin:
If we are marked to die, we are enough
To do our country loss; and if to live,
The fewer men, the greater share of honour.

图8-6 HTML的输出显示

i 输入输出和标准库。另一个有用的结构是*BinIO*，它支持8位字节形式的二进制数据的输入输出。字符和字节并不是一回事：在有些系统中，字符不止8位，并且对于某些字符码有特别的解释。例如，二进制输入输出没用换行的记法。

函子*ImperativeIO*、*StreamIO*和*PrimIO*支持更低级的输入输出。（库将它们描述为可选的，但是好一点的ML系统都会提供。）*ImperativeIO*支持带缓冲的命令式操作。*StreamIO*提供了函数式的输入操作：读入项并不从输入流中移除，而是产生新的输入流。*PrimIO*是最原始的一层，没有缓冲，直接以操作系统调用的方式实现。通过应用这些函子可以支持专门类型的输入输出，例如扩展字符集。

Andrew Appel在John Reppy和Dave Berry的帮助下设计了这个输入输出接口。

348
350

练习 8.30 书写一个ML程序来对一个文件中包含的行、单词和字符进行计数。单词是一个以空格、制表符或换行分隔的字符串。

练习 8.31 书写一个过程，提示输入一个圆的半径，打印出相应的圆面积（利用公式 $A = \pi r^2$ ），并不断重复这一过程。如果对（输入的）实数进行解释的尝试失败，程序应该打印一条错误信息并让用户重试。

练习 8.32 `<` `>` `&` `"` 这四个字符在HTML中有特殊的含义。它们在输入中的出现应当被替换成相应的转义序列 `<` `>` `&` `"`。修改*htmlCvt*以完成这个要求。

8.10 美化打印程序

在显示程序和数学公式时，如果通过分行和缩进来强调它们的结构，那么程序和公式会更加易读。4.19节的重言式检测器包括一个函数*show*，它能将命题转换成字符串，如果转换后的字符串过长，在一行内写不下，那么我们通常会看到这样的输出（30字符一行的情况下）：

```
(((((landed | saintly) | ((~land
ded) | (~saintly))) & (((~rich
) | saintly) | ((~landed) | (~
saintly)))) & (((landed | rich
) | ((~landed) | (~saintly)))
& (((~rich) | rich) | ((~lande
d) | (~saintly)))))
```

图8-7的显示就好多了，它是由一个美化打印程序处理后输出的。两个命题（包括上面那个）分别格式化成30和60个字符一行。找出公式的最佳表现形式需要判断力和审美观，不过一个简单的美化打印模式能给出意想不到的好结果。有些ML系统提供了类似下面讲述的那种美化打印原语。

美化打印程序接受一段文字，这些文字需要由嵌套信息来修饰，并允许插入断点。让我们用尖括号（`<`）指示嵌套，用竖线（`|`）指示可能分行的位置。形如 $\langle e_1 \cdots e_n \rangle$ 的表达式称为一块（block）。

例如，表达式块

$$\left(\left(a * \mid b \right) - \mid \left(\left(c + \mid d \right) \right) \right)$$

351

表示了字符串 $a*b-(c+d)$ ，它允许在字符 $*$ 、 $-$ 和 $+$ 后面换行。

```

((~(((~landed) | rich) &
  (~(saintly & rich)))) |
  (~landed) | (~saintly)))

((((landed | saintly) |
  ((~landed) | (~saintly))) &
  (((~rich) | saintly) |
  ((~landed) |
    (~saintly)))) &
  (((landed | rich) |
    ((~landed) | (~saintly))) &
    (((~rich) | rich) |
      (~landed) | (~saintly))))))

((~(((~landed) | rich) & (~(saintly & rich)))) |
  (~landed) | (~saintly)))

((((landed | saintly) | ((~landed) | (~saintly))) &
  (((~rich) | saintly) | ((~landed) | (~saintly)))) &
  (((landed | rich) | ((~landed) | (~saintly))) &
    (((~rich) | rich) | ((~landed) | (~saintly))))))

```

图8-7 美化打印程序的输出

当根据运算符的优先级省去括号时，有必要正确地处理美化打印。表达式块的嵌套结构对应于公式

$$(a \times b) - (c + d) \quad \text{而不是} \quad a \times (b - (c + d))$$

如果 $a*b-(c+d)$ 在一行内写不下，那么它应该在 $-$ 字符后面断行，外层块要先于内层块进行断行。

美化打印算法跟踪当前行剩下的空格数。当它遇到断点时，它要确定在同一块或仅外层块中的下个断点前有多少个字符。（这样就会忽略内层块中的断点。）如果当前行写不下那么多字符，算法则会另起一行，并适当缩进以和当前块首对齐。

算法并不要求在每块之后都立即跟随一个断点。在上例中，表达式块

$$\left(c + \quad \mid \quad d \right)$$

紧跟的是一个 $)$ 字符，所以字符串 $d)$ 不能断开。因此要以某种方式确定到下一断点的距离。

美化打印程序具有如下签名

```

signature PRETTY =
  sig
    type t
    val blo : int * t list -> t
    val str : string -> t
    val brk : int -> t
    val pr  : TextIO.outstream * t * int -> unit
  end;

```

并提供比刚才所述原语的花样略多一些:

- t 是符号表达式的类型, 也就是所说的块、字符串和断点。
- $blo(i, [e_1, \dots, e_n])$ 创建一个包含已知各表达式的块, 并指定当前的缩进要增加 i 。这个缩进信息将在块被断开时用到。
- $str(s)$ 创建一个包含字符串 s 的表达式。
- $brk(l)$ 创建一个长度为 l 的断点, 如果不需要在此断行, 则换成打印 l 个空格。
- $pr(os, e, m)$ 打印表达式 e 到输出流 os 上, 以 m 作为行长。

352
353

图8-8给出了美化打印程序。注意 *Block* 里存有由 *blo* 计算出来的整个块的大小。另外, *after* 里放着从当前块尾到下一个断点的距离。

```
structure Pretty : PRETTY =
  struct
    datatype t = Block of t list * int * int
              | String of string
              | Break of int;

    fun breakdist (Block(_,_,len)::es, after) = len + breakdist (es, after)
      | breakdist (String s :: es, after)      = size s + breakdist (es, after)
      | breakdist (Break _ :: es, after)       = 0
      | breakdist ([], after)                  = after;

    fun pr (os, e, margin) =
      let val space = ref margin

          fun blanks n = (TextIO.output(os, StringCvt.padLeft #" " n "");
                        space := !space - n)

          fun newline () = (TextIO.output(os, "\n"); space := margin)

          fun printing ([], _, _) = ()
            | printing (e::es, blockspace, after) =
              (case e of
                 Block(bes, indent, len) =>
                   printing(bes, !space-indent, breakdist (es, after))
              | String s => (TextIO.output(os, s); space := !space - size s)
              | Break len =>
                   if len + breakdist (es, after) <= !space
                   then blanks len
                   else (newline(); blanks(margin-blockspace));
                   printing (es, blockspace, after))

            in printing([e], margin, 0); newline() end;

    fun length (Block(_,_,len)) = len
      | length (String s)      = size s
      | length (Break len)     = len;

    val str = String and brk = Break;

    fun blo (indent, es) =
      let fun sum ([], k) = k
          | sum (e::es, k) = sum(es, length e + k)
          in Block(es, indent, sum(es, 0)) end;
      end;
```

图8-8 美化打印程序

图8-7所显示的输出是由我们的重言式检测器经由如下修正后生成的:

```
local open Pretty
in
  fun prettyshow (Atom a)      = str a
  | prettyshow (Neg p)         =
      blo(1, [str("~", prettyshow p, str")"])
  | prettyshow (Conj(p,q))     =
      blo(1, [str(" ", prettyshow p, str" &",
                  brk 1, prettyshow q, str")"])
  | prettyshow (Disj(p,q))     =
      blo(1, [str(" ", prettyshow p, str" |",
                  brk 1, prettyshow q, str")"]);

end;
> val prettyshow = fn : prop -> Pretty.t
```

以prettyshow的结果作为参数调用Pretty.pr将完成美化打印。

❶ 进一步的阅读。美化打印程序由Oppen (1980) 发起。Oppen的算法很复杂, 但只需要很少的空间; 它可以处理非常大的文件, 但只需缓存几行的文本。我们的美化打印程序足以显示定理和其他易于在内存中存储的计算结果。Kennedy (1996) 给出了一个绘制树的ML程序。

练习 8.33 举例说明形如

$$\left\langle \left\langle E_1 * \mid E_2 \right\rangle - \mid \left\langle (\left\langle E_3 + \mid E_4 \right\rangle) \right\rangle \right\rangle$$

的块怎样才有可能被美化打印成在*字符后断行, 而不是在-字符后面断行的。这个问题有多严重? 给出修改算法的建议以纠正这一问题。

练习 8.34 实现一种新块, 含有“一致断点”: 除非当前行可以写下整块内容, 否则所有断点都要换行。例如, 一致断行下面的语句

$$\left\langle \text{if } E \mid \text{then } E_1 \mid \text{else } E_2 \right\rangle$$

将产生 $\text{if } E$
 then E_1 而不会产生 $\text{if } E \text{ then } E_1$
 else E_2 else E_2

练习 8.35 书写一个纯函数式版本的美化打印程序。它必须返回一个字符串列表, 而不是写入一个流。函数式的版本有什么实际优点?

练习 8.36 Fortran语句

```
FORMAT (' Input =', I6, ' Output =', F8.2)
```

描述了一行文本, 它以字符串'Input ='开始, 后跟一个6位字符的整数, 再跟字符串'Output=', 最后跟着一个占用了8个字符的浮点(实)数, 其中两个数字在小数点后面。一个用Fortran格式写入的文件可以以同样的格式读出。讨论这种类型的格式化输入输出能否

在ML中实现。怎样来表示格式和数据？

要点小结

- 引用表示内存中可变更的单元，类似过程式语言中的变量和指针。
- 在ML中，变量不能被更新，只有引用和数组可以被更新。
- 为了防止多态引用导致运行错误，在多态的`val`声明中出现的表达式必须是语法值。
- 循环数据结构，例如环形缓冲区，可以利用引用来构造。
- 一个函数可以利用命令式特性，同时表现为纯函数式的特性。
- 输入和输出命令在程序和外部设备间传递字符。

第9章 书写 λ -演算的解释器

本章综合了到目前为止所学的全部概念。作为一个扩展的例子，这里给出了一系列模块来将 λ -演算实现为一个原始的函数式程序设计语言。 λ -演算的项可以被分析、求值并显示结果。当然，这个语言几乎不具备实用性，简单的算术运算使用的是一元记法并需要数分钟才能完成计算。然而，它的实现涉及了很多的基本技术：语法分析、约束变量的表示以及将表达式归约到范式。这些技术可以应用到定理证明和计算机代数上。

本章提要

我们将讨论语法分析和两个 λ -项解释器，并回顾一下 λ -演算。本章包含以下几节：

- 函数式语法分析器。一个ML函子实现了自顶向下的递归下降语法分析。分析器可以通过中缀操作符组合在一起，这些中缀操作符类似于将语法短语组合起来的符号。
- λ -演算简介。这种演算中的项可以表示函数式程序。它们可以通过传值调用或传名调用的机制进行求值。替换的进行需要十分小心，以避免变量名字的冲突。
- 在ML中表示 λ -项。替换、语法分析和美化打印将作为ML的结构来实现。
- 作为程序设计语言的 λ -演算。包括无穷表在内的典型的函数式语言数据结构将在 λ -演算中进行编码。这里还将演示递归函数的求值。

函数式语法分析器

在讨论 λ -演算之前，我们先看看怎样书写函数式的扫描器和语法分析器。下面讲述的语法分析器是对上一章美化打印程序的补充。通过这些工具，ML程序可以读写 λ -项、ML类型和逻辑公式。

357

9.1 扫描或词法分析

语法分析器基本不会直接去操作字符串。字符首先被扫描 (scan)：处理成为词法单元 (token)，例如关键字、标识符、特殊符号和数字。语法分析器将接受一个词法单元列表。

这样的两层方案简化了用于分析的文法。扫描器以统一的方式删除了空格、换行和注释，让语法分析器去处理语法中更为复杂的部分。扫描可以借助一个有限状态机来进行，这种状态机由以字符为索引的数组驱动，可以运行得非常快。对于小规模输入，库结构 *Substring* 中的扫描函数足以胜任了。

词法分析器是一个具有如下签名的结构：

```
signature LEXICAL =  
  sig  
    datatype token = Id of string | Key of string  
    val scan : string -> token list  
  end;
```

词法单元是一个标识符或关键字。这个简单的扫描器不能识别数字。调用`scan`将对一个字符串进行词法分析，并返回词法单元的结果列表。

在对语言进行语法分析之前，我们必须描述它的词汇。为了将词法单元归类为标识符或关键字，必须给扫描器提供一个签名`KEYWORD`的实例：

```
signature KEYWORD =
  sig
    val alphas : string list
    and symbols : string list
  end;
```

表`alphas`定义了字母数字组成的关键字，像`"if"`和`"let"`，而`symbols`则列出了符号组成的关键字，像`"("`和`)"`。这两种关键字要分别处理：

- 由字母数字组成的字符串要尽可能长地扫描，直到后面跟随的不再是字母或数字为止。当它属于`alphas`时，就被归类为关键字，否则将归类为标识符。
- 由符号字符组成的字符串要扫描到它匹配`symbols`中的某个元素，或者后面跟随的不再是符号字符为止。它总是被归类为关键字。例如，如果`"("`属于`symbols`，那么`"(("`将被扫描成两个`"("`词法单元，如果不属于`symbols`则作为一个`"(("`词法单元。

函子`Lexical`（图9-1）利用几个`Substring`中的函数实现了这个扫描器，这些函数包括：`getc`、`splitl`、`string`、`dropl`和`all`。函数`getc`将一个子串分成它的首字符和后续字符两部分，这两部分组成了一个序偶，如果该子串为空，那么返回的结果是`NONE`。在8.9节，我们见过函数`all`和`string`，它们进行字符串和子串之间的相互转换。我们也见过`splitl`，它从左到右扫描子串，将子串分成两部分。函数`dropl`与此类似，但只是返回子串的第二部分，扫描器利用它来略过空格和其他非显示字符。库谓词`Char.isAlphaNum`用来识别字母和数字，`Char.is Graph`用来识别所有的输出字符，而`Char.isPunct`是用来识别标点符号的。

这段代码是直接且高效的，速度可以与常见的命令式实现相媲美。虽然`Substring`中函数的表现是函数式的，但是它们以递增索引的方式工作。这比处理字符列表要好。

该函子声明了函数`member`作为内部使用。它不依赖于第3章声明的中缀操作符`mem`，也不依赖于任何标准库之外的其他顶层函数。这个成员测试专用于类型`string`，因为多态的相等测试会比较慢。

将词法分析器实现为一个函子是因为签名`KEYWORD`中的信息是静态的。在对新的语言进行语法分析的时候，我们只需要改变关键字表和特殊符号表即可。将函子应用到某个`KEYWORD`的实例上相当于将其中的信息包装在结果结构中。我们当然也可以将词法分析器实现为一个柯里函数，以类似的信息作为一个记录参数，但是这样做会使词法分析器的类型变得复杂，换来的只是没有必要的灵活性。

练习 9.1 修改这个扫描器来识别输入中的十进制数。定义一个新的构造子`Num : integer → token`来返回扫描进来的整数常量值。

练习 9.2 修改这个扫描器来略过注释。括住注释的符号，例如`"(*"和"*)"`，应该作为组件增添到结构`Keyword`中。

```

functor Lexical (Keyword: KEYWORD) : LEXICAL =
  struct
    datatype token = Key of string | Id of string;

    fun member (x:string, l) = List.exists (fn y => x=y) l;

    fun alphaTok a =
      if member(a, Keyword.alphas) then Key(a) else Id(a);

    (* 扫描符号关键字 *)
    fun symbolic (sy, ss) =
      case Substring.getc ss of
        NONE          => (Key sy, ss)
      | SOME (c, ssl) =>
          if member(sy, Keyword.symbols)
          orelse not (Char.isPunct c)
          then (Key sy, ss)
          else symbolic (sy ^ String.str c, ssl);

    (* 将一个子串扫描称为一个词法单元表 *)
    fun scanning (toks, ss) =
      case Substring.getc ss of
        NONE          => rev toks      (* 子串结束 *)
      | SOME (c, ssl) =>
          if Char.isAlphaNum c
          then (* 标识符或关键字 *)
              let val (id, ss2) = Substring.splitl Char.isAlphaNum ss
                  val tok       = alphaTok (Substring.string id)
              in scanning (tok::toks, ss2)
              end
          else if Char.isPunct c
          then (* 特殊符号 *)
              let val (tok, ss2) = symbolic (String.str c, ssl)
              in scanning (tok::toks, ss2)
              end
          else (* 忽略空格、换行、控制字符 *)
              scanning (toks, Substring.dropl (not o Char.isGraph) ss);

    fun scan a = scanning([], Substring.all a);
  end;

```

图9-1 词法分析函子

9.2 自顶向下的语法分析套件

自顶向下的递归下降语法分析器非常像它所分析的文法本身。对每个语法短语都有过程与之对应，而这些过程的相互递归调用精确地反映了那些文法规则。

这种相像在函数式程序设计中更为突出。高阶函数可以表示类似短语连接、选择短语和短语重复这样的语法操作。在对中缀操作符作出适当的挑选后，函数式的语法分析器可以编写得和一套文法规则几乎完全一样。但是不要被表象所迷惑，这个程序具有自顶向下语法分析的所有局限性，具体来说，一个左递归的 (left-recursive) 的文法规则如

$$exp = exp \text{ " * "}$$

会使该语法分析器进入死循环! 编译原理的课本上有关于处理这些局限的讲述。

方案提要。假设文法包含了某一类短语, 这类短语的含义可以由具有类型 τ 的值表示。这些短语的语法分析器一定是具有类型

$$token\ list \rightarrow \tau \times token\ list$$

的函数, 此后将缩写为类型 $\tau\ phrase$ 。当给语法分析器一个以有效短语开始的词法单元列表时, 它将移除那些词法单元, 并计算出它们的含义, 作为一个类型 τ 的值。语法分析器返回这个含义和剩下的词法单元所组成的序偶。如果词法单元表没有以有效的短语开始, 那么语法分析器会以抛出异常`SyntaxErr`的方式拒绝这个表。

并不是所有具有类型 $\tau\ phrase$ 的函数都是语法分析器。语法分析器只能从词法单元表的前端移除词法单元, 它不应插入词法单元, 或是以其他方式修改词法单元表。

为了实现复杂的语法分析器, 我们将定义一些基本的语法分析器和一些用以组合语法分析器的操作。

分析基本短语。普通的语法分析器能识别标识符, 特定的关键字, 或空短语。它们从输入中最多能移除一个词法单元:

- 语法分析器`id`, 类型为`string phrase`, 从输入中移除一个`Id`词法单元, 并将这个标识符作为字符串返回 (和词法单元表的尾部配成一个序偶)。
- 如果`a`是一个字符串, 则语法分析器`$a`具有类型`string phrase`。它从输入中移除关键字词法单元`Key a`, 并返回`a`和词法单元表尾组成的序偶。
- 语法分析器`empty`具有多态类型 $(\alpha\ list)\ phrase$ 。它返回`[]`和最初的词法单元表组成的序偶。

头两个语法分析器会拒绝没有以所需词法单元开始的输入, 而`empty`总是成功的。

选择短语。如果`ph1`和`ph2`都具有类型 $\tau\ phrase$, 那么`ph1 || ph2`也具有这个类型。语法分析器`ph1 || ph2`接受那些被语法分析器`ph1`或`ph2`所接受的短语。当给予一个词法单元表时, 这个语法分析器将其传递给`ph1`, 如果成功则返回它的结果, 如果`ph1`拒绝这个词法单元则尝试`ph2`。

语法分析器`!!ph`和`ph`是一样的, 只是在`ph`拒绝词法单元时会令整个分析失败, 并产生一个错误信息。这可以防止外层的`||`操作符去尝试另一种短语分析。操作符`!!`通常用于以特别的关键字开始的短语, 因此没有其他可选择的分析方法, 另见下面的`$--`。

连贯短语。语法分析器`ph1 -- ph2`接受一个`ph1`短语且其后紧跟一个`ph2`短语。这个语法分析器传递给定的词法单元表到`ph1`。如果`ph1`分析出一个短语并返回 $(x, toks2)$, 则剩下的词法单元 $(toks2)$ 会被传递给`ph2`。如果`ph2`也分析出一个短语并返回 $(y, toks3)$, 则`ph1 -- ph2`返回 $((x, y), toks3)$ 。注意, `toks3`包含了两次分析后剩下的词法单元。如果任一分析器拒绝了它的输入, 则`ph1 -- ph2`也是。

如此一来, `ph1 -- ph2`的含义将是`ph1`和`ph2`含义的序偶, 它适用于输入中连贯在一起的两段。如果`ph1`具有类型 $\tau_1\ phrase$, `ph2`具有类型 $\tau_2\ phrase$, 那么`ph1 -- ph2`则具有类型 $(\tau_1 \times \tau_2)\ phrase$ 。

操作符`$--`则涵盖了一个常用的情形。语法分析器`a $-- ph`和`$a -- ph`类似, 它分析一个

以关键词法单元 a 开始的短语，并继续调用 ph 。但是它只返回 ph 的含义，并不将其与 a 组成序偶。还有，如果 ph 拒绝了它收到的词法单元，那么（利用 $!!$ ）这个语法分析器会导致整个分析失败并产生一个错误信息。操作符 $\$--$ 适用于仅有一个文法规则是以符号 a 开始的情形。

修改含义。语法分析器 $ph \gg f$ 接受与 ph 相同的输入，但是当 ph 返回 $(x, toks)$ 时，它返回 $(f(x), toks)$ 。这样一来，在 ph 赋予含义 x 时，它赋予含义 $f(x)$ 。如果 ph 具有类型 $\sigma \text{ phrase}$ 且 f 具有类型 $\sigma \rightarrow \tau$ ，则 $ph \gg f$ 具有类型 $\tau \text{ phrase}$ 。


重复。为了说明这些操作符，让我们来编写一个分析算子。如果 ph 是任意一个语法分析器，那么 $repeat\ ph$ 将不做分析或重复地使用 ph 进行分析：

```
fun repeat ph toks = (   ph -- repeat ph >> (op::)
                        || empty   ) toks;
```

362

中缀操作符 $--$ 、 $>>$ 、 $||$ 的优先级是从高到低的。 $repeat$ 的函数体由两个语法分析器构成，它们经 $||$ 连接，和显而易见的文法定义相似： ph 的重复或者是一个 ph 接一个 ph 的重复，或者是空。

语法分析器 $ph -- repeat\ ph$ 返回 $((x, xs), toks)$ ，其中 xs 是一个表。操作符 $>>$ 应用表的“构造”（操作符 $::$ ）将序偶 (x, xs) 转换为 $x :: xs$ 。在第二行， $empty$ 产生 $[]$ 作为空短语的含义。简而言之， $repeat\ ph$ 构造一个重复短语的含义列表。如果 ph 具有类型 $\tau \text{ phrase}$ ，则 $repeat\ ph$ 具有类型 $(\tau \text{ list}) \text{ phrase}$ 。

 小心无穷递归。 $repeat$ 的声明能否通过在两边同时省略 $toks$ 而简化？答案是否定的——调用 $repeat\ ph$ 会立即产生一个对 $repeat\ ph$ 的递归调用，这是灾难性的：

```
fun repeat ph toks = ph -- repeat ph >> (op::) || empty;
```

使用形式参数 $toks$ 是延迟对 $repeat$ 函数体求值的一种办法，求值将在得到作为实际参数的词法单元表之后进行，正常的话，内层的 $repeat\ ph$ 将被给予一个减短了的词法单元表，因此可以终止。惰性求值将不需要这种办法。

9.3 语法分析器的ML代码

关于操作符 $\$--$ 、 $--$ 、 $>>$ 和 $||$ 的中缀指令给它们赋予了适当的优先级。确切的数字是随意选择的，但是 $\$--$ 的优先级必须比 $--$ 高，以便吸收它左边的字符串。另外， $>>$ 的优先级必须比 $--$ 低，以便它可以包括整个文法规则。最后， $||$ 必须具有最低优先级，使得它可以将文法规则组合起来。

```
infix 6 $--;
infix 5 --;
infix 3 >>;
infix 0 ||;
```

这些指令具有全局效果，因为它们是在顶层使用的。我们还应该打开（open）含有分析操作符的结构：复合名字不能用作中缀操作符。

函子Parsing（图9-3）实现了这个语法分析器。函子的声明使用了基本形式，只接受一个结构作为参数，此处具体为 Lex 。它的结果签名是 $PARSE$ （图9-2）。


```
signature PARSE =
  sig
    exception SyntaxErr of string
    type token
    val id      : token list -> string * token list
    val $       : string -> token list -> string * token list
    val empty   : 'a -> 'b list * 'a
    val ||      : ('a -> 'b) * ('a -> 'b) -> 'a -> 'b
    val !!      : ('a -> 'b * 'c) -> 'a -> 'b * 'c
    val --      : ('a -> 'b * 'c) * ('c -> 'd * 'e) -> 'a -> ('b * 'd) * 'e
    val $--     : string * (token list -> 'a * 'b) -> token list -> 'a * 'b
    val >>      : ('a -> 'b * 'c) * ('b -> 'd) -> 'a -> 'd * 'c
    val repeat  : ('a -> 'b * 'a) -> 'a -> 'b list * 'a
    val infixes :
      (token list -> 'a * token list) * (string -> int) *
      (string -> 'a -> 'a -> 'a) -> token list -> 'a * token list
    val reader  : (token list -> 'a * 'b list) -> string -> 'a
  end;
```

图9-2 函数式语法分析器的签名

你可能会注意到这个签名中的许多类型和上一节给出的不同。类型缩写

$$\alpha \text{ phrase} = \text{token list} \rightarrow \alpha \times \text{token list}$$

并没有被用到，更重要的是，签名中的某些类型比语法分析所需要的更具一般性，它们并不局限于词法单元表。

ML经常会赋予一个函数比我们预想的要更为多态的类型。如果在编写函数之前描述签名——这是一种规矩的软件开发模式——则会失去额外的多态性。在设计签名时，有时在ML顶层进行查询，看看它所给出的类型是什么也是有帮助的。

签名PARSE描述了类型token，以便进一步描述id和其他项的类型。相应地，Parsing将类型token声明为和Lex.token等价。

函数reader将一个语法分析器包装起来给外部使用。调用reader ph a将字符串a扫描成词法单元，并将它们提供给语法分析函数ph。如果再没有词法单元剩下，那么reader就返回短语的含义，否则它将提示发生了语法错误。

分析中缀操作符。函数infixes构造了一个语法分析器来对中缀操作符进行分析，它有以下参数：

- ph接受准备通过操作符组合的原子短语。
- prec_of给出操作符的优先级，对于不是中缀操作符的关键字一律返回-1。
- apply将短语的含义进行组合，apply a x y将操作符a应用到操作数x和y上。

作为结果的语法分析器将识别如下的输入

$$ph \oplus ph \otimes ph \ominus ph \oslash ph$$

并根据操作符的优先级来结合原子短语。它利用了相互递归的函数over和next。

调用over k将分析一系列短语，它们被优先级大于或等于k的操作符隔开。在next k (x, toks)中参数x是前一个短语的含义，而k则是指标优先级。这个调用所做的是，若下一词法单

元是具有优先级 k 或以上的操作符 a ，诸词法单元将被递归地由 $over(prec_of\ a)$ 进行分析，并且它们的结果要和 x 组合起来，然后这一结果以及剩下的词法单元以原来的优先级 k 为指标被进一步分析；若下一词法单元不满足上述条件，则调用什么也不做。

```

functor Parsing (Lex: LEXICAL) : PARSE =
  struct
    type token = Lex.token;

    exception SyntaxErr of string;

    fun id (Lex.Id a :: toks) = (a, toks)
      | id toks                = raise SyntaxErr "Identifier expected";

    fun $a (Lex.Key b :: toks) = if a=b then (a, toks)
                                  else raise SyntaxErr a
      | $a _                  = raise SyntaxErr "Symbol expected";

    fun empty toks = ([], toks);

    fun (ph1 || ph2) toks = ph1 toks handle SyntaxErr _ => ph2 toks;

    fun !! ph toks = ph toks handle SyntaxErr msg =>
      raise Fail ("Syntax error: " ^ msg);

    fun (ph1 -- ph2) toks =
      let val (x, toks2) = ph1 toks
          val (y, toks3) = ph2 toks2
      in ((x, y), toks3) end;

    fun (ph>>f) toks =
      let val (x, toks2) = ph toks
      in (f x, toks2) end;

    fun (a $-- ph) = ($a -- !!ph >> #2);

    fun repeat ph toks = ( ph -- repeat ph >> (op::)
                          || empty ) toks;

    fun infixes (ph, prec_of, apply) =
      let fun over k toks = next k (ph toks)
          and next k (x, Lex.Key(a)::toks) =
            if prec_of a < k then (x, Lex.Key a :: toks)
            else next k ((over (prec_of a) >> apply a x) toks)
          | next k (x, toks) = (x, toks)
      in over 0 end;

    (* 扫描和语法分析，检查是否没有多余的词法单元 *)
    fun reader ph a =
      (case ph (Lex.scan a) of
        (x, []) => x
       | (_, _::_) => raise SyntaxErr "Extra characters in phrase");

  end;

```

图9-3 语法分析函子

这个算法不处理括号，括号应由 ph 处理。10.6节演示了 $infixes$ 的使用。

书写回溯的语法分析器。当某个词法单元表不止有一种语法描述时，语法是有歧义的。我们的方法要易于修改以使得每个分析函数都能返回一个成功的结果序列（惰性表）。检查序列中的元素以使回溯能在输入的全部语法描述上进行。

语法分析器 $ph1 -- ph2$ 用于对紧跟着 $ph2$ 的 $ph1$ 进行语法分析，并返回一个由所有可能的

分析方法所组成的序列。它将 $ph1$ 应用到输入上,得到一个 $(x, toks2)$ 序偶的序列。对该序列中的每一个元素,语法分析器都将 $ph2$ 应用到 $toks2$ 上,得到一个 $(y, toks3)$ 序偶的序列。最后它将返回所有成功结果 $((x, y), toks3)$ 的序列。对每个结果,含义 (x, y) 是由 $ph1$ 和 $ph2$ 所返回的结果构成的序偶。

语法分析器返回一个空序列来表示对其输入的拒绝,而不是抛出异常。注意,如果 $ph1$ 拒绝其输入,或是 $ph2$ 拒绝 $ph1$ 的所有结果,那么 $ph1 \dashv\vdash ph2$ 将产生空序列,也就是拒绝它的输入。

这是一个很有意思的有关序列处理的练习,但是它具有回溯语法分析器的弱点:速度慢,错误处理能力差。它可能需要指数级的时间来分析输入,自底向上的语法分析则要快得多。如果输入含有语法错误,回溯语法分析器除了返回空序列以外没有任何信息。我们的语法分析器稍加修改就能够对语法错误进行精确的定位。修改类型 $token$,使得每个词法单元都带有它在输入字符串中的位置,并且让!!在它的错误报告中包含这一信息。

回溯在定理证明中是有价值的。寻找证明的一个“策略”可以表达为一个以目标为参数,并返回解序列的函数。策略可以组合起来形成有效的搜索过程。下一章将展示这一技术,它和我们处理语法分析函数的方式有关。

练习 9.3 给出语法分析器 ph 的一个例子,使得对于所有输入, ph 都能成功地完成语法分析,而 $repeat\ ph$ 则是死循环。

练习 9.4 语法分析树(parse tree)是一棵树,它表示了词法单元表在分析之后的结构。每个结点都代表一个短语,它的分支是构成短语的符号和子短语。修改我们的分析方法使得它能构造语法分析树。声明一个合适的分析树类型 $partree$,使每个语法分析函数都可具有类型

$$token\ list \rightarrow partree \times token\ list$$

编写操作符 $||$ 、 $--$ 、 id 、 $\$$ 、 $empty$ 和 $repeat$,注意, $>>$ 不再具有任何作用。

练习 9.5 修改语法分析方法以产生成功结果的序列,就像上面叙述的那样。

练习 9.6 以过程式风格编写语法分析方法,其中每个分析“函数”都具有类型 $unit \rightarrow \alpha$ 并通过删除词法单元来更新指向词法单元表的引用。过程式方案有缺点吗?还是比函数式方案要好?

练习 9.7 修改签名 $PARSE$ 来描述一个具有签名 $LEXICAL$ 的子结构 Lex ,取代描述类型 $token$,使得其他签名项可以引用类型 $Lex.token$ 。相应地修改函子声明。

练习 9.8 当一个表达式包含多个具有相同优先级的中缀操作符时, $infixes$ 是将它们左结合还是右结合?修改该函数以实现相反的结合。叙述一个算法来处理左结合和右结合操作符的混合情况。

9.4 例子:分析和显示类型

语法分析器和美化打印程序将通过ML的类型文法进行演示。出于举例的目的,ML的类型系统可以通过放弃记录和元组类型来得以简化。有两种形式的类型要考虑:

由类型构造子应用到零个或多个类型参数上而构成的类型,如 int 、 $bool\ list$ 和 $(\alpha\ list) \rightarrow (\beta\ list)$ 。这里,类型构造子 int 被应用到零个参数上, $list$ 被应用到类型 $bool$ 上,而 \rightarrow 则被应用到了类型 $\alpha\ list$ 和 $\beta\ list$ 上。ML对大多数类型构造子采用后缀语法,但是 \rightarrow 则具有中缀语法。在

内部，这些类型可以被表示成一个字符串和一个类型表所组成的序偶。

一个类型可以仅由一个类型变量构成，这个类型变量由一个字符串来表示。我们的类型结构具有如下签名：

```
signature TYPE =
  sig
    datatype t = Con of string * t list | Var of string
    val pr    : t -> unit
    val read  : string -> t
  end;
```

它描述了三个组件：

- 数据类型 *t* 包含两种形式的类型，类型构造子 *Con* 和类型变量 *Var*。
- 调用 *pr ty* 将在终端上打印类型 *ty*。
- 函数 *read* 将一个字符串转换成一个类型。

我们可以通过函子来实现这个签名，其中函子的参数表会包含签名 *PARSE* 和 *PRETTY*。但是，一般来说避免书写函子会简单一些，除非要多次用到函子。因此让我们给词法分析器和语法分析器创建结构来进行类型的语法分析。稍后将用它们来实现λ-演算。

结构 *LamKey* 定义了必需的符号。结构 *LamLex* 提供了针对于类型和λ-演算的词法分析，而 *LamParsing* 则提供了语法分析操作符。

```
structure LamKey =
  struct val alphas = []
        and symbols = ["(", ")", "'", "->", "'"]
  end;
structure LamLex  = Lexical (LamKey);
structure LamParsing = Parsing (LamLex);
```

结构 *Type* (图9-4) 匹配签名 *TYPE*。为了简单起见，它只处理了 \rightarrow 符号，其他的类型构造子将留作练习。文法相互递归地定义了类型和原子类型。一个原子类型或者是一个类型变量，或者是括号括住的任何类型：

$$\begin{aligned} \text{Type} &= \text{Atom} \rightarrow \text{Type} \\ &| \text{Atom} \\ \text{Atom} &= ' \text{Id} \\ &| (\text{Type}) \end{aligned}$$

这个文法将 \rightarrow 看成是一个中缀操作符，并且向右结合。它将 'a \rightarrow 'b \rightarrow 'c 解释为 'a \rightarrow ('b \rightarrow 'c)，而不是 ('a \rightarrow 'b) \rightarrow 'c，因为 'a \rightarrow 'b 不是一个原子 (*Atom*)。

结构里含有两个 *local* 声明，一个是关于语法分析的，另一个是关于美化打印的。每个都声明了相互递归的函数 *typ* 和 *atom*，这与文法相对应。打开结构 *LamParsing* 使得它的操作在顶层可用，要知道，中缀指令是全局的。

类型的语法分析。实际上，在使用了自顶向下的语法分析操作符之后，语法分析器中的函数定义和文法规则是完全一样的。操作符 \gg 出现了三次，它将一个函数应用到了语法分析器的返回结果上。函数 *typ* 通过 \gg 将 *makeFun* 应用到第一条文法规则的结果上，将两个类型合并成一个函数类型。在规则中使用 $\$--$ 可以避免将箭头符号作为短语的组成部分返回。

```

structure Type : TYPE =
  struct

    datatype t = Con of string * t list
              | Var of string;

    local (** 语法分析**)
      fun makeFun (ty1,ty2) = Con(">=", [ty1,ty2]);
      open LamParsing

      fun typ toks =
        (
          atom -- ">=" $-- typ          >> makeFun
          || atom
        ) toks
      and atom toks =
        (
          $"'" -- id                    >> (Var o op^)
          || "(" $-- typ -- $"")       >> #1
        ) toks;

    in
      val read = reader typ;
    end;

    local (** 显示**)
      fun typ (Var a)                  = Pretty.str a
        | typ (Con(">=", [ty1,ty2])) = Pretty.blo(0, [atom ty1,
                                                    Pretty.str ">=",
                                                    Pretty.brk 1,
                                                    typ ty2])

      and atom (Var a) = Pretty.str a
        | atom ty      = Pretty.blo(1, [Pretty.str "(",
                                         typ ty,
                                         Pretty.str ")"]);

    in
      fun pr ty = Pretty.pr (TextIO.stdOut, typ ty, 50)
    end
  end;
end;

```

图9-4 分析和显示ML类型

*atom*的两个情形都涉及到>>,以及两个神秘的函数。对于第一种情形,在分析类型变量'a期间,>>将`Var o op^`应用到了序偶("'", "a")上。这个函数由`Var`和字符串连接函数复合而成,它将字符串连接成"'a"并返回类型`Var "'a"`。

在*atom*的第二个情形中,分析短语(*Type*)将调用函数#1,它选择其参数的第一个分量。这里它接受(`ty, "("`)作为参数并返回`ty`。如果我们没有使用`$--`来分析左括号,那么我们将需要用到更加神秘的函数(`#2 o #1`)。

语法分析程序使用参数*toks*来避免死循环(就像上面的*repeat*一样),这也因为`fun`声明必须要有一个参数。

类型的美化打印。像语法分析中的那样,同样的相互递归对显示也有效。函数`typ`和*atom*都会把类型转换为一个符号表达式以用于美化打印,不过*atom*都将它的结果括在括号中,除非结果只是一个标识符。括号应当仅在需要时出现,括号太多反而令人困惑。

结构*Pretty*中的函数*blo*、*str*和*brk*以典型的方式来描述块、字符串和断点。函数*atom*调用*blo*，并指定一个字符的缩进以将后续的换行对齐在左括号之后。函数*typ*调用*blo*时指定零缩进，因为它不包含任何括号，在字符串"*->*"之后，它调用*brk* 1来设定一个空格或换行。

函数*pr*输出到终端（输出流*TextIO.stdout*），使用50作为行宽。

尝试一些例子。我们可以输入一些类型，注意它们在语法分析之后的内部表示（作为*Type.t*的值），并检查它们的显示是否正确：

```
Type.read "a->'b->'c";
> Con ("->", [Var "'a",
>           Con ("->", [Var "'b", Var "'c"])]])
> : Type.t
Type.pr it;
> 'a -> 'b -> 'c
Type.read "('a->'b)->'c";
> Con ("->", [Con ("->", [Var "'a", Var "'b"])]),
>           Var "'c"])
> : Type.t
Type.pr it;
> ('a -> 'b) -> 'c
```

我们对于类型的语法分析是朴素的。形如（*Type*）的字符串一定要被分析两次。*Type*的第一条文法规则没有生效：在右括号之后不存在词法单元*->*。第二条文法规则成功地将它分析成一个原子。我们是修改文法来去除原子的重复出现。

❶ 有关语法分析的更多信息。LR语法分析是复杂文法所选用的分析方法，例如程序设计语言的那些文法。这种自底向上的技术可靠、高效并且具有普遍性，它也支持很好的错误恢复。LR语法分析器不是手工书写的，而是通过使用像Yacc（yet another compiler-compiler，意为：另一个编译器生成器）这样的工具生成的。这个工具接受一个文法，构造出语法分析表并将语法分析器以源代码的形式输出。每条语法规则都可附以一个语义动作（semantic action）：只要规则适用就执行代码。大多数语法分析器生成程序都是基于C语言的。

ML-Yacc（Tarditi和Appel，1994）使用ML描述语义动作以及生成出来的语法分析器。ML-Yacc的设置相当复杂，但是对于一定规模的文法还是值得考虑的。你必须为ML-Yacc提供一个词法分析器，这既可以通过手工书写，也可以使用像ML-Lex（Appel等，1994）的工具生成。

自顶向下进行语法分析的函数式方案已经为人所知很长时间了。Burge（1975）中有一份最早发表的相关描述，包括使用惰性表来进行回溯。Reade（1989）给出了一份更为新式的叙述。Frost和Launchbury（1989）为一个解答系统而使用这种方法来对英语的一个子集进行语法分析。曾经建议了!!和\$--符号的Tobias Nipkow使用这个方案来分析Isabelle理论文件。

Aho等（1986）相当好的描述了词法分析和语法分析。它既涵盖了这里所实现自顶向下方案，也包括了以ML-Yacc为基础的自底向上方案。

练习 9.9 实现任意类型构造子的语法分析和美化打印。首先为ML的后缀语法定义一条文法，像下面这样

```
'c list list          (string,int) sum
('a -> 'b) list      'a list -> 'b list
```

当类型构造子只有一个参数并且不涉及箭头时，括号是可选的，由此，`'a->'b list`代表`'a->(('b) list)`，而不是`('a->'b) list`。

练习 9.10 使用语法分析原语，或使用ML-Yacc，实现命题——4.17节的类型`prop`——的语法分析器。

λ-演算简介

图灵机、递归函数和寄存器机器都是计算的形式模型。λ-演算是由丘奇 (Alonzo Church) 开发的，它是最早的计算模型之一，也许还是最现实的。λ-演算可以表示序偶、表、树（甚至是无穷的）和高阶函数的计算。大多数函数式语言只不过是λ-演算的一种修饰过的形式，它们的实现也是以λ-演算理论为基础的。

丘奇命题 (Church's thesis) 断言有效可计算函数恰好是那些可以在λ-演算中计算的函数。因为“有效”是一个模糊的概念，所以丘奇命题无法被证明，但是已知λ-演算和其他计算模型具有相同的计算能力。在这些模型中编写的函数，只要有足够的空间和时间，都可以有效地计算，尚未发现任何可计算函数不能在这些模型下编写。

9.5 λ-项和λ-归约

λ-演算是一种关于函数的简单形式理论。它里面的项称为λ-项，是递归地从变量 x, y, z, \dots 和其他λ-项中构造出来的。令 t, u, \dots 代表任意的λ-项，它们可以具有以下三种形式：

x	一个变量
$(\lambda x.t)$	函数抽象 (abstraction)
(tu)	函数应用 (application)

372

若项 t_1 含在项 t_2 中或和它相等，则 t_1 是 t_2 的子项 (subterm)。例如， y 是 $(\lambda z.(zy))$ 的子项。

在抽象 $(\lambda x.t)$ 中，我们称 x 为约束变量 (bound variable)， t 为函数体 (body)。在 t 中的每个 x 都被抽象所约束。相对地，如果有一个变量 y 没有被约束，即如果它没有被包含在某种抽象 $(\lambda y.u)$ 的函数体内，则它是自由的 (free) 的。例如，在 $(\lambda z.(\lambda x.(y x)))$ 中 x 是约束的，而 y 是自由的。从此以后，用 a, b, c, \dots 表示自由变量。

约束变量的名字是什么并没有多大意义。如果它们在抽象中被一致改名，那么新的抽象和原来的基本上是一样的。这个原则在数学中是广为人知的。在积分 $\int_a^b f(x)dx$ 中，变量 a 和 b 是自由的，而 x 是约束的。在乘积 $\prod_{k=0}^n p(k)$ 中，变量 n 是自由的，而 k 是约束的。

抽象 $(\lambda x.t)$ 表示函数 f ，含义是对于所有 x ，有 $f(x) = t$ 。将 $(\lambda x.t)$ 应用到参数 u 上将产生一项，它是将 t 中所有自由出现的 x 替换成 u 而生成的。我们把这一替换的结果写成 $t[u/x]$ 。替换涉及很多细微之处，不过这些留待后面再讲。

λ-转换。这是一些在保持λ-项直观含义的同时对它们进行变换的规则。转换不应和像 $x + y = y + x$ 这样的等式混淆，等式是关于已知算术运算的陈述。λ-演算并不关心已有的数学对象。λ-项本身是对象，而λ-转换则是基于它们的符号变换。

最重要的是β-转换，它通过将实际参数替换进函数体来变换一个函数应用：

$$((\lambda x.t)u) \Rightarrow_{\beta} t[u/x]$$

下面这个例子中，实际参数是($g\ a$):

$$((\lambda x.((f\ x)x))(g\ a)) \Rightarrow_{\beta} ((f(g\ a))(g\ a))$$

下面是一个连续两次β-转换的例子:

$$((\lambda z.(z\ a))(\lambda x.x)) \Rightarrow_{\beta} ((\lambda x.x)\ a) \Rightarrow_{\beta} a$$

α-转换将一个抽象中的约束变量重新命名:

$$(\lambda x.t) \Rightarrow_{\alpha} (\lambda y.t[y/x])$$

x 上的抽象被变换为 y 上的抽象，并且 x 被替换成 y 。例如:

$$\begin{aligned} (\lambda x.a\ x) &\Rightarrow_{\alpha} (\lambda y.a\ y) \\ (\lambda x.(x(\lambda y.(y\ x)))) &\Rightarrow_{\alpha} (\lambda z.(z(\lambda y.(y\ z)))) \end{aligned}$$

373

两个λ-项如果可以通过α-转换（有可能应用到子项）由一个变换到另一个，那么它们是全等的（congruent）。直观上，我们可以认为全等的项都是一样的，在需要的地方随时对约束变量重新命名。然而，自由变量的名字是有意义的，因此 a 和 b 是不同的，而 $(\lambda x.x)$ 和 $(\lambda y.y)$ 是全等的。

记法。嵌套的抽象和应用可以简写:

$$\begin{aligned} (\lambda x_1.(\lambda x_2. \dots (\lambda x_n.t) \dots)) &\text{可写作 } (\lambda x_1 x_2 \dots x_n.t) \\ (\dots (t_1\ t_2) \dots t_n) &\text{可写作 } (t_1\ t_2 \dots t_n) \end{aligned}$$

当一项不包含在另一项之中，或者是一个抽象的函数体时，外面的括号可以省略。例如，

$$(\lambda x.(x(\lambda y.(y\ x)))) \text{可写作 } \lambda x.x(\lambda y.y\ x)$$

归约到范式。归约步骤（reduction step） $t \Rightarrow u$ 通过对 t 的任何一个子项应用β-转换将 t 变换到 u 。如果一项不再允许进一步的归约，那么它就是一个范式（normal form）。规范（normalize）一项意味着不断应用归约直到最终结果是一个范式。

有些项可以通过多种途径归约。丘奇-罗瑟（Church-Rosser）定理说明了从同一项开始的不同归约序列最终将汇合到一起。特别地，不存在两个归约序列会得出不同的（不全等的）范式。项的范式可以看成是它的值，它和归约进行的顺序无关。

例如， $(\lambda x.a\ x)((\lambda y.b\ y)\ c)$ 有两个不同的归约序列，这两个序列都有相同的范式。每步受影响的子项都以下划线标出:

$$\begin{aligned} (\lambda x.a\ x)((\lambda y.b\ y)\ c) &\Rightarrow a((\lambda y.b\ y)\ c) \Rightarrow a(bc) \\ (\lambda x.a\ x)((\lambda y.b\ y)\ c) &\Rightarrow (\lambda x.a\ x)(bc) \Rightarrow a(bc) \end{aligned}$$

有很多λ-项是没有范式的。例如， $(\lambda x.x\ x)(\lambda x.x\ x)$ 用β-转换会变换回自身。任何想规范这一项的尝试必然不能终止:

$$(\lambda x.x\ x)(\lambda x.x\ x) \Rightarrow (\lambda x.x\ x)(\lambda x.x\ x) \Rightarrow \dots$$

374

项 t 即使在某些归约序列不能终止的情况下也可以有范式。典型情况是， t 包含没有范式的子项

u ，而 u 却在某个归约步骤中被删除了。例如，归约序列

$$(\lambda y.a)((\lambda x.xx)(\lambda x.xx)) \Rightarrow a$$

通过删除项 $(\lambda x.xx)(\lambda x.xx)$ 直接得到了范式。这对应于函数的传名调用：实际参数不被求值，而是直接替换进函数体。试图规范实际参数的话将产生一个不可终止的归约序列：

$$(\lambda y.a)((\lambda x.xx)(\lambda x.xx)) \Rightarrow (\lambda y.a)((\lambda x.xx)(\lambda x.xx)) \Rightarrow \dots$$

在替换进函数体之前先对实际参数求值就是对函数应用进行传值调用。在上面的例子中，应用传值调用策略是得不到范式的。如果范式存在，那么对应于传名调用的归约策略将总能得到该范式。

你很可能要问， $\lambda x.xx$ 怎么会是函数？它可以应用于任何对象上，并将该对象应用于自身！在传统的数学中，一个函数只能在已有值的集合上定义。 λ -演算处理函数的方式和传统的理解不同。^①

9.6 在替换中防止变量的捕获

必须小心地定义替换：否则转换可能会出错。例如，项 $\lambda x.y.y x$ 应该表现得像柯里函数一样，当应用到参数 t 和 u 上时，返回 $u t$ 作为结果。对于所有的 λ -项 t 和 u ，我们都应有归约

$$(\lambda x.y.y x)t u \Rightarrow (\lambda y.y t)u \Rightarrow u t$$

而下面的归约序列显然是错的：

$$(\lambda x.y.y x)y b \Rightarrow (\lambda y.y y)b \Rightarrow b b \quad ???$$

从 $(\lambda x.y.y x) y$ 到 $\lambda y.y y$ 的 β -转换是不正确的，因为自由变量 y 变成约束的了。这一替换捕获(capture)了该自由变量。通过预先将约束变量 y 重新命名为 z ，归约就可以安全地进行：

$$(\lambda x.z.z x)y b \Rightarrow (\lambda z.z y)b \Rightarrow b y$$

375

一般来说，只要 u 中的自由变量不是 t 中的约束变量，替换 $t[u/x]$ 就不会捕获任何变量。

如果约束变量由文字表示，那么替换有时必须要重新命名 t 中的约束变量以避免捕获自由变量。重新命名是复杂的，并且可能损失效率。我们必须保证新名字不会在项的其他地方出现。一般都喜欢把新名字起成和旧名字相似的，像G6620094这样的变量名字是不为人乐见的。

无名表示法。改变 λ -项的表示可以简化替换算法。一个约束变量名字 x 的作用只是将每个 x 和约束它的 λx 匹配起来，以便归约可以正确进行。如果这些匹配可以通过其他办法完成，那么名字就是可以取消的。

我们可以利用抽象的嵌套深度来做到这一点。每个约束变量都由一个索引表示，索引给出的是它和约束它的抽象之间的抽象数目。两个 λ -项是全等的（只是在 α -转换下不同）当且仅当它们的无名表示相等。

在无名记法中， λ 符号后面不跟变量名，约束变量索引以数的形式出现。在 $\lambda x.(\lambda y.x) x$ 的

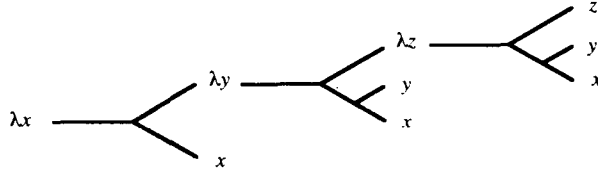
① Dana Scott构造过一个模型，其中所有的抽象，包括 $\lambda x.x x$ ，都代表一个函数。(Barendregt, 1984)。然而，本章仅从纯语法的角度看待 λ -演算。

函数体中第一次出现的 x 由1表示，因为它被包含在基于 y 的抽象中。第二个 x 没有包括在任何其他的抽象中，故表示为0。因此， $\lambda x.(\lambda y.x) x$ 的无名表示是 $\lambda.(\lambda.1) 0$ 。

下面这一项的约束变量出现在多个嵌套深度中：

$$\lambda x.x (\lambda y.x y (\lambda z.x y z))$$

将这一项看成一棵树可以突出它的嵌套结构：



在无名记法中，三个 x 分别被表示为0、1和2：

$$\lambda.0 (\lambda.1 0 (\lambda.2 1 0))$$

像抽象和替换这样的操作在无名表示法中很容易进行。对于变量绑定来说，这是个不错的数据结构，但却不是一种可读的记法。原有的变量名应该保留以备后用，以便使用者可以看到传统的记法。

376

抽象。假设 t 是一个λ-项，我们准备把它构造造成基于所有自由变量 x 的抽象 $\lambda x.t$ 。例如，我们将 $x (\lambda y.a x y)$ 作为这样的一项，它的无名记法是

$$x (\lambda.a x 0)$$

要约束所有的 x ，我们必须将它们替换成正确的索引值，这里分别是0和1，然后插入λ符号：

$$\lambda.0 (\lambda.a 1 0)$$

这可以通过一个以项为参数的递归函数来完成，此递归函数将会对抽象的深度进行计数。每个 x 都被替换成与其深度相等的索引。

替换。为了进行β-转换

$$(\lambda x.t)u \Rightarrow_{\beta} t[u/x]$$

必须对项 t 进行递归变换，用 u 替换掉所有的 x 。在无名记法中， x 可能会被表示为多个不同的索引。索引一开始为0，随着 t 中抽象深度的递增。例如，转换

$$(\lambda x.x (\lambda y.a x y))b \Rightarrow_{\beta} b (\lambda y.a b y)$$

在无名记法中变成

$$(\lambda.0 (\lambda.a 1 0))b \Rightarrow_{\beta} b (\lambda.a b 0)$$

我们会看到， x 在外层抽象中具有索引0，而在内层抽象中具有索引1。

对子项 $(\lambda x.t) u$ 进行β-转换更为复杂。实际参数 u 可能含有在外层被约束的变量，也就是在 u 中没有抽象与之相匹配的索引。这些索引必须加上当前的嵌套深度，然后才能替换到 t 中去，这保证了它们之后能引用相同的抽象。

例如，在

$$\lambda z.(\lambda x.x (\lambda y.x))(a z) \Rightarrow_{\beta} \lambda z.a z (\lambda y.a z)$$

377

中，有两处要被替换成实际参数 a z ，其中一个处于 λy 的作用域内。在无名方案中， a z 将得到两种不同的表示：

$$\lambda.(\lambda.0(\lambda.1))(a\ 0) \Rightarrow_{\beta} \lambda.a\ 0(\lambda.a\ 1)$$

练习 9.11 规范下面的项，并给出所有的归约序列

$$(\lambda f.f(f\ a))((\lambda x.x\ x)((\lambda y.y)(\lambda y.y)))$$

练习 9.12 给出下面每一项的范式，或说明其没有范式：

$$\begin{aligned} &(\lambda f\ x\ y.f\ x\ y)(\lambda u\ v.u) \\ &(\lambda x.f(x\ x))(\lambda x.f(x\ x)) \\ &(\lambda x\ y.y\ x)(\lambda x.f(f\ x))(\lambda x.f(f(f\ x))) \\ &(\lambda x.x\ x)(\lambda x.x) \end{aligned}$$

练习 9.13 给出下列项的无名表示：

$$\begin{aligned} &\lambda x\ y\ z.x\ z(y\ z) \\ &\lambda x\ y.(\lambda z.x\ y\ z)y\ x \\ &\lambda f.(\lambda x.f(\lambda y.x\ x\ y))(\lambda x.f(\lambda y.x\ x\ y)) \\ &(\lambda p\ x\ y.p\ x\ y)(\lambda x\ y.y)\ a\ b \end{aligned}$$

练习 9.14 给出一种 λ -项的表示法，在内部用唯一的整数来标明约束变量。给出构造 λ -项和进行替换的算法。

在ML中表示 λ -项

基于无名表示法，在ML中可以直接实现 λ -演算。下一节将给出抽象和替换的ML程序，并将对 λ -项进行语法分析和美化打印。

我们需要在7.10节中声明的字典结构`StringDict`。它允许我们将任意信息和字符串关联起来，这里的信息是指 λ -项。我们可以基于一个已定义标识符的环境（environment）来对 λ -项进行求值。

9.7 基本操作

378

下面是无名表示法的签名：

```
signature LAMBDA =
sig
  datatype t = Free of string
              | Bound of int
              | Abs of string * t
              | Apply of t * t;

  val abstract : int -> string -> t -> t
  val absList  : string list * t -> t
  val applyList : t * t list -> t
  val subst    : int -> t -> t -> t
```

```
val inst      : t StringDict.t -> t -> t
end;
```

数据类型 t 由自由变量（作为字符串）、约束变量（作为索引）、抽象和应用构成。每个Abs结点都存储了约束变量的名字以用于显示打印。

调用 $abstract\ i\ b\ t$ 将 t 中所有的自由变量 b 都转换成索引 i （或一个更大的索引，一般出现在嵌套抽象中）。通常 $i = 0$ ，并且结果将马上被包在一个抽象中以匹配这个索引。对 t 里面的抽象进行递归调用会有 $i > 0$ 的情况。

调用 $absList([x_1, \dots, x_n], t)$ 将建立抽象 $\lambda x_1 \dots x_n. t$ 。

调用 $applyList(t, [u_1, \dots, u_n])$ 将建立应用 $t\ u_1 \dots u_n$ 。

调用 $subst\ i\ u\ t$ 将 t 中具有索引 i 的约束变量替换成 u 。通常 $i = 0$ ，且 t 是 β -转换 $(\lambda x. t)\ u$ 中的抽象函数体。对 t 中的抽象进行递归调用时会有 $i > 0$ 的情况。所有超过 i 的索引都要减一，以对该索引的删除进行调整。

调用 $inst\ env\ t$ 将复制 t ，并将其中所有在 env 中有定义的变量都替换成它们的定义。字典 env 表示一个环境， $inst$ 则展开了项中所有的定义。这一过程称为实例化（instantiation）。定义中可能引用了其他定义，实例化将持续到结果中不再出现已定义的变量。

签名LAMBDA是具体的，暴露了所有的内部细节。许多类型 t 的值都是假的（improper）：它们不对应任何真实的 λ -项，这是由于它们包含了没有匹配项的约束变量索引。对于任何 i 都不存在一个可以表示为Bound i 的（真）项。此外， $abstract$ 返回假项，而 $subst$ 正好需要它们。抽象的 λ -演算签名会提供基于 λ -项自身的操作，而隐藏它们的表示法。

结构Lambda（图9-5）实现了这个签名。函数 $shift$ 是私有的，因为它只被 $subst$ 所调用。调用 $shift\ i\ d\ u$ 将 i 加到 u 中所有满足 $j > d$ 的没有匹配的索引 j 上。起初 $d = 0$ ，接着 d 将在对 u 中的抽象进行递归调用时递增。在将某项 u 替换进另一项之前，任何 u 中未匹配的索引都需要转变。

379

```
structure Lambda : LAMBDA =
  struct
    datatype t = Free of string
              | Bound of int
              | Abs of string * t
              | Apply of t * t;

    (* 在项中将b转换为约束索引i *)
    fun abstract i b (Free a)      = if a=b then Bound i else Free a
      | abstract i b (Bound j)     = Bound j
      | abstract i b (Abs(a,t))    = Abs(a, abstract (i+1) b t)
      | abstract i b (Apply(t,u)) = Apply(abstract i b t, abstract i b u);

    (* 多个自由变量的抽象 *)
    fun absList (bs,t) = foldr (fn (b,u) => Abs(b, abstract 0 b u)) t bs;

    (* t在多个项上的应用 *)
    fun applyList (t0,us) = foldl (fn (u,t) => Apply(t,u)) t0 us;
```

图9-5 λ-项的无名表示法

```

(* 将项中的非局部索引上移i *)
fun shift 0 d u          = u
  | shift i d (Free a)    = Free a
  | shift i d (Bound j)   = if j>=d then Bound(j+i) else Bound j
  | shift i d (Abs(a,t))  = Abs(a, shift i (d+1) t)
  | shift i d (Apply(t,u)) = Apply(shift i d t, shift i d u);

(* 在项t中用u替换约束变量i *)
fun subst i u (Free a)    = Free a
  | subst i u (Bound j)   =
      if j<i then Bound j (* 局部约束 *)
      else if j=i then shift i 0 u
      else (*j>i*) Bound(j-1) (* 非局部于t *)
  | subst i u (Abs(a,t))  = Abs(a, subst (i+1) u t)
  | subst i u (Apply(t1,t2)) = Apply(subst i u t1, subst i u t2);

(* 替换自由变量 *)
fun inst env (Free a)      = (inst env (StringDict.lookup(env,a))
                             handle StringDict.E _ => Free a)
  | inst env (Bound i)     = Bound i
  | inst env (Abs(a,t))    = Abs(a, inst env t)
  | inst env (Apply(t1,t2)) = Apply(inst env t1, inst env t2);
end;

```

图9-5 (续)

函数`inst`只替换自由变量，而不替换约束变量。它需要接受真 λ -项，即其中不存在没有匹配的索引。因此，它不需要跟踪嵌套的深度，也不需要调用`shift`。

练习 9.15 解释在`absList`和`applyList`声明中的折叠算子是怎样使用的。

练习 9.16 为 λ -演算声明一个签名，隐藏它的内部表示法。它需要描述谓词以测试一个 λ -项是不是一个变量、抽象或应用，并描述一些函数来进行抽象和替换。勾勒出两个具有这一签名的结构设计，利用两种不同的 λ -项表示法。

9.8 λ -项的语法分析

为了可以应用语法分析器和美化打印程序，需要为 λ -项定义一个文法，其中包括嵌套抽象和应用的缩写形式。下面的文法将一般项和原子项做了区分。我们使用百分号(%)来代表 λ 符号：

$$\begin{aligned} \text{Term} &= \% \text{ Id Id}^* \text{ Term} \\ &\quad | \text{ Atom Atom}^* \end{aligned}$$

$$\begin{aligned} \text{Atom} &= \text{Id} \\ &\quad | (\text{ Term }) \end{aligned}$$

注意， phrase^* 代表零个或多个 phrase 的重复。由多个原子排在一起组成的项，例如 $a\ b\ c\ d$ ，是嵌套应用(((a b) c) d)的缩写。更为自然的文法是定一个短语类`Applic`：

$$\begin{aligned} \text{Applic} &= \text{Atom} \\ &\quad | \text{ Applic Atom} \end{aligned}$$

然后就可以将Term里面的Atom Atom*换成Applic。但是Applic的第二条语法规则是左递归的，它会使得我们的语法分析器进入死循环。前面的文法就是通过标准做法消除左递归之后的产物。

结构ParseTerm（图9-6）用到结构Parse和Lambda，包括了语法分析器和λ-项的操作，以满足签名PARSE_TERM:

```
signature PARSE_TERM =
  sig val read: string -> Lambda.t end;
```

这个结构的唯一目的就是対λ-项进行语法分析。它的签名只描述了一个组件：函数read，它将一个字符串转换成一个λ-项。它的实现是很直接的，利用了结构Lambda的组件absList和applyList。

```
structure ParseTerm : PARSE_TERM =
  struct
    fun makeLambda ((b, bs), t) = Lambda.absList (b::bs, t);
    open LamParsing
    fun term toks =
      (   "%" $-- id -- repeat id -- "." $-- term >> makeLambda
        || atom -- repeat atom                                >> Lambda.applyList
      ) toks
    and atom toks =
      (   id                                >> Lambda.Free
        || "(" $-- term -- ")"             >> #1
      ) toks;
    val read = reader term;
  end;
```

图9-6 λ-演算的语法分析器

练习 9.17 在函数makeLambda中，为什么它的参数具有那样的模式？

练习 9.18 对"%x x.x(%x x.x)"进行语法分析的结果是什么？

9.9 显示λ-项

结构DisplayTerm（图9-7）实现了λ-项的美化打印。它使用了结构Pretty和Lambda（美化打印程序和项操作），并满足签名DISPLAY_TERM:

```
signature DISPLAY_TERM =
  sig
    val rename : string list * string -> string
    val stripAbs : Lambda.t -> string list * Lambda.t
    val pr      : Lambda.t -> unit
  end;
```

这个签名描述了多个组件:

- rename([a_1, \dots, a_n], a) (在必要时) 将撇号 (') 追加在a之后使它和每个 a_1, \dots, a_n 区别开来。

- *stripAbs* 将一个抽象分解为它的约束变量和函数体，我们会在下面作进一步讲述。
- 调用 *pr t* 将项 *t* 打印在终端上。

即便是使用无名表示法，约束变量在显示项时也可能需要重新命名。范式 $(\lambda xy.x)$ *y* 显示为 $\%y'.y$ ，而不是 $\%y.y$ 。函数 *stripAbs* 和它的辅助函数 *strip* 是处理抽象的。给定 $\lambda x_1 \dots x_m.t$ ，约束变量将被重新命名以区别于 *t* 中的所有自由变量。新的名字将替换到 *t* 中去作为自由变量。这样一来项中所有索引在显示时就都被消去了。

```
structure DisplayTerm : DISPLAY_TERM =
  struct
    (* 项中的自由变量 *)
    fun vars (Lambda.Free a)      = [a]
      | vars (Lambda.Bound i)     = []
      | vars (Lambda.Abs(a,t))    = vars t
      | vars (Lambda.Apply(t1,t2)) = vars t1 @ vars t2;

    (* 重新命名变量"a"以避免冲突 *)
    fun rename (bs,a) =
      if List.exists (fn x => x=a) bs then rename (bs, a ^ "'") else a;

    (* 删除前导lambda; 返回约束变量名 *)
    fun strip (bs, Lambda.Abs(a,t)) =
      let val b = rename (vars t, a)
      in strip (b::bs, Lambda.subst 0 (Lambda.Free b) t)
      end
      | strip (bs, u)                = (rev bs, u);

    fun stripAbs t = strip ([],t);

    fun spaceJoin (b,z) = " " ^ b ^ z;

    fun term (Lambda.Free a)      = Pretty.str a
      | term (Lambda.Bound i)     = Pretty.str "??UNMATCHED INDEX??"
      | term (t as Lambda.Abs _) =
      let val (b::bs,u) = stripAbs t
          val binder    = "%" ^ b ^ (foldr spaceJoin ". " bs)
      in Pretty.blo(0, [Pretty.str binder, term u])
      end
      | term t                  = Pretty.blo(0, applic t)
    and applic (Lambda.Apply(t,u)) = applic t @ [Pretty.brk 1, atom u]
      | applic t                    = [atom t]
    and atom (Lambda.Free a)      = Pretty.str a
      | atom t                    = Pretty.blo(1, [Pretty.str "(",
                                                  term t,
                                                  Pretty.str ")"]);

    fun pr t = Pretty.pr (TextIO.stdOut, term t, 50);
  end;
```

图9-7 λ -演算的美化打印程序

相互递归的函数 *term*、*applic* 和 *atom* 将 λ -项准备好以便进行美化打印。自由 (*Free*) 变量直接显示其名字。约束 (*Bound*) 变量索引不应出现，除非它没有对应的 *Abs* 结点（说明这是一个假项）。对于 *Abs* 结点，约束变量被重新命名，然后通过 *foldleft* 将其连成一个字符串，以空格隔开。*Apply* 结点通过 *applic* 来显示，它对应于上一节提到的文法短语 *Applic*。最后，*atom* 要括在括号中，单独的标识符除外。

练习 9.19 $(\lambda x y.x) (\lambda y.y)$ 的范式将如何显示? 修改`DisplayTerm`以保证当显示一项时, 在重叠的作用域中不存在被约束两次的变量名。

练习 9.20 不用自由变量来替换约束变量也可以显示项。修改`DisplayTerm`来保存变量约束列表, 其中变量约束位于包含当前子项的各抽象中。在显示项`Bound i`时, 在该表中找到第*i*个名字。

作为程序设计语言的λ-演算

虽然λ-演算简单, 然而对于建造完整的函数式程序设计模型来说已经足够丰富了。像序偶和表这样的数据结构可以在传值调用或传名调用的求值策略下进行处理。在对这些问题进行简单的讨论后, 我们将用ML来进行说明。首先必须做出一些定义。

如果*t*可以经由零次或多次归约步骤变换成*u*, 那么就记为 $t \Rightarrow^* u$ 。如果*u*是范式, 那么 $t \Rightarrow^* u$ 可以看成是对*t*求值得到结果*u*。并不是所有的求值策略都可以成功地找到这个范式。

如果存在某项*u* (不一定是范式!) 使得 $t_1 \Rightarrow^* u$ 以及 $t_2 \Rightarrow^* u$, 那么就记为 $t_1 = t_2$ 。如果 $t_1 = t_2$, 那么一旦范式存在, 两者就具有同一范式。在将范式看作值的情况下, $t_1 = t_2$ 意味着 t_1 和 t_2 具有相同的值。

我们书写 $a \equiv t$ 来表示“将*a*定义为*t*的缩写”, 其中*a*是自由变量。

9.10 λ-演算中的数据结构

现在来考虑如何对布尔值、序偶、自然数和表编码。下面给出的编码是随意的, 真正重要的是这些数据结构及其操作满足某些标准性质。对于布尔类型的编码必须将真值`true`和`false`, 以及条件操作符`if`定义为λ-项, 并 (对于所有*t*和*u*) 满足

$$\begin{aligned} \text{if } \text{true } t \ u &= t \\ \text{if } \text{false } t \ u &= u \end{aligned}$$

一旦有了两个不同的真值和条件操作符, 我们就可以定义否定、合取和析取。类似地, ML编译器可以用任意的位模式来表示`true`和`false`, 只要操作表现正确即可。

布尔值。布尔值可以通过如下编码来定义

$$\begin{aligned} \text{true} &\equiv \lambda x y.x \\ \text{false} &\equiv \lambda x y.y \\ \text{if} &\equiv \lambda p x y.p \ x \ y \end{aligned}$$

所需性质很容易被验证, 例如:

$$\begin{aligned} \text{if } \text{true } t \ u &\equiv (\lambda p x y.p \ x \ y) \text{true } t \ u \\ &\Rightarrow (\lambda x y.\text{true } x \ y) t \ u \\ &\Rightarrow (\lambda y.\text{true } t \ y) u \\ &\Rightarrow \text{true } t \ u \\ &\Rightarrow (\lambda x y.x) t \ u \\ &\Rightarrow (\lambda y.t) u \\ &\Rightarrow t \end{aligned}$$

这就建立了 $\text{if } \text{true } t \ u \Rightarrow^* t$, 因此有 $\text{if } \text{true } t \ u = t$ 。

序偶。编码必须描述函数 $pair$ （用于构造序偶的）以及投影函数 fst 和 snd （用于选择序偶的分量）。通常的编码是

$$pair \equiv \lambda x y f.f x y$$

$$fst \equiv \lambda p.p \ true$$

$$snd \equiv \lambda p.p \ false$$

其中， $true$ 和 $false$ 如前定义。下面的归约和所对应的等式对于所有 t 和 u 都成立，这一点很容易验证：

$$fst(pair\ t\ u) \Rightarrow^* t$$

$$snd(pair\ t\ u) \Rightarrow^* u$$

自然数。在几个已知的自然数编码中，丘奇的编码是最典雅的。带下划线的数字 $\underline{0}, \underline{1}, \dots$ ，代表丘奇数（church numeral）：

$$\underline{0} \equiv \lambda f\ x.x$$

$$\underline{1} \equiv \lambda f\ x.f\ x$$

$$\underline{2} \equiv \lambda f\ x.f(f\ x)$$

$$\vdots$$

$$\underline{n} \equiv \lambda f\ x.f^n(x)$$

这里 $f^n(x)$ 是 $\underbrace{f(\dots(f\ x)\dots)}_{n\text{次}}$ 的缩写。

函数 suc 用于计算一个数的后继， $iszero$ 用于测试一个数是否为零：

$$suc \equiv \lambda n f\ x.n\ f\ (f\ x)$$

$$iszero \equiv \lambda n.n(\lambda x.false)true$$

下面的归约不难验证，其中 \underline{n} 是任意丘奇数：

$$suc\ \underline{n} \Rightarrow^* \underline{n+1}$$

$$iszero\ \underline{0} \Rightarrow^* true$$

$$iszero(suc\ \underline{n}) \Rightarrow^* false$$

386

丘奇数允许定义简洁得出奇的加法、乘法和指数运算：

$$add \equiv \lambda m\ n\ f\ x.m\ f(n\ f\ x)$$

$$mult \equiv \lambda m\ n\ f.m(n\ f)$$

$$expt \equiv \lambda m\ n\ f\ x.n\ m\ f\ x$$

这些都可以用归纳法来形式地验证，它们背后的直观意义也是简单的。每个丘奇数 \underline{n} 都是一个算子，它将一个函数应用 n 次。我们看到

$$add\ \underline{m}\ \underline{n}\ f\ x = f^m(f^n(x)) = f^{m+n}(x),$$

其他的定义也可以类似地理解。

自然数编码还必须描述一个前驱函数 pre ，使得对于所有数 n 有

$$pre(suc\ n) = n$$

对于丘奇数来说，从 $n+1$ 计算出 n 是复杂的（也很慢！），给定 f 和 x ，我们必须找到某个 g 和 y ，从 $g^{n+1}(y)$ 中计算出 $f^n(x)$ 。一个合适的 g 是一个作用在序偶上的函数，它使得 $g(z, z') = (f(z), z)$ 对所有 (z, z') 都成立，则有

$$g^{n+1}(x, x) = (f^{n+1}(x), f^n(x))$$

我们提取其中的第二个分量。为了将这些形式化，我们定义 $prefn$ 来构造 g 。然后定义前驱函数 pre 和减法函数 sub ：

$$\begin{aligned} prefn &\equiv \lambda f\ p.pair(f(fst\ p))\ (fst\ p) \\ pre &\equiv \lambda n\ f\ x.snd(n(prefn\ f)(pair\ x\ x)) \\ sub &\equiv \lambda m\ n.n\ pre\ m \end{aligned}$$

关于减法， $sub\ m\ n = pre^n(m)$ ，它计算了 m 的第 n 个前驱。

表。表是用序偶和布尔值来编码的。一个以 x 为首元素， y 为表尾的非空表编码为 $(false, (x, y))$ 。空表 nil 可以被编码为 $(true, true)$ ，然而下面的这个定义更为简单有效：

$$\begin{aligned} nil &\equiv \lambda z.z \\ cons &\equiv \lambda x\ y.pair\ false\ (pair\ x\ y) \\ null &\equiv fst \\ hd &\equiv \lambda z.fst(snd\ z) \\ tl &\equiv \lambda z.snd(snd\ z) \end{aligned}$$

387

易于对任意 t 和 u 检查定义的基本性质：

$$\begin{aligned} null\ nil &\Rightarrow^* true \\ null(cons\ t\ u) &\Rightarrow^* false \\ hd(cons\ t\ u) &\Rightarrow^* t \\ tl(cons\ t\ u) &\Rightarrow^* u \end{aligned}$$

传名调用不用对 u 求值就可将 $hd(cons\ t\ u)$ 简化为 t ，并且可以处理无穷表。

练习 9.21 基于任意的布尔值编码，定义一种序偶的编码。通过将布尔值编码为 $true = \lambda x\ y.y$ 以及 $false = \lambda x\ y.x$ 来演示你的定义。

练习 9.22 对于任意的丘奇数 m 和 n ，验证：

$$\begin{aligned} iszero(suc\ n) &= false \\ add\ m\ n &= m + n \\ mult\ m\ n &= m \times n \\ expt\ m\ n &= m^n \end{aligned}$$

练习 9.23 定义一种自然数编码, 使得它具有简单的前驱函数。

练习 9.24 定义一种带标签的二叉树编码。

练习 9.25 书写一个ML函数 *numeral*, 它具有类型 $int \rightarrow \text{Lambda.t}$, 使得对于所有 $n \geq 0$, *numeral* n 都能构造出丘奇数 n 。

9.11 λ -演算中的递归定义

存在一个通过递归计算丘奇数阶乘的 λ -项 *fact*

$$\text{fact } n = \text{if } (\text{iszero } n) \text{ } \underline{1} \text{ } (\text{mult } n \text{ } (\text{fact}(\text{pre } n)))$$

存在一个通过递归合并两个表的 λ -项 *append*

$$\text{append } z \text{ } w = \text{if } (\text{null } z) \text{ } w \text{ } (\text{cons}(\text{hd } z)(\text{append}(\text{tl } z)w))$$

还存在一个满足递归方程的 λ -项 *inflist*

388

$$\text{inflist} = \text{cons MORE inflist}$$

它是无穷表 $[MORE, MORE, \dots]$ 的编码。

递归定义是借助于 λ -项 *Y* 来编码的:

$$Y \equiv \lambda f. (\lambda x. f(x \ x))(\lambda x. f(x \ x))$$

虽然 *Y* 背后的构思是晦涩难懂的, 但是一个简单的计算就可以验证 *Y* 对于所有 λ -项 *f* 满足不动点性质 (fixed point property)

$$Y f = f(Y f)$$

我们可以利用这一性质来不断地展开递归对象的函数体。定义

$$\begin{aligned} \text{fact} &\equiv Y(\lambda g \text{ } n. \text{if } (\text{iszero } n) \text{ } \underline{1} \text{ } (\text{mult } n \text{ } (g(\text{pre } n)))) \\ \text{append} &\equiv Y(\lambda g \text{ } z \text{ } w. \text{if } (\text{null } z) \text{ } w \text{ } (\text{cons}(\text{hd } z)(g(\text{tl } z)w))) \\ \text{inflist} &\equiv Y(\lambda g. \text{cons MORE } g) \end{aligned}$$

在每个定义中, 出现的递归都由 $Y(\lambda g. \dots)$ 中的约束变量 *g* 来替换。我们来验证 *inflist* 的递归方程, 其他的与此类似。第一行和第三行由定义可知成立, 而第二行则用到了不动点性质:

$$\begin{aligned} \text{inflist} &\equiv Y(\lambda g. \text{cons MORE } g) \\ &= (\lambda g. \text{cons MORE } g)(Y(\lambda g. \text{cons MORE } g)) \\ &\equiv (\lambda g. \text{cons MORE } g)\text{inflist} \\ &= \text{cons MORE inflist} \end{aligned}$$

用 *Y* 编写的递归函数在传名调用归约下可以正确地执行。如要使用传值调用归约, 则递归函数必须使用另一种不动点算子 (我们将在下面讨论) 来编写, 否则执行将无法终止。

9.12 λ -项的求值

结构 *Reduce* (图9-8) 实现了传名调用和传值调用的归约策略。它的签名是 *REDUCE*:

```
signature REDUCE =
sig
  val eval      : Lambda.t -> Lambda.t
  val byValue   : Lambda.t -> Lambda.t
  val headNF    : Lambda.t -> Lambda.t
  val byName    : Lambda.t -> Lambda.t
end;
```

389

签名描述了四个求值函数：

- *eval* 使用类似ML的传值调用策略对项进行求值。它的结果不必是范式。
- *byValue* 用传值调用来规范项。
- *headNF* 将项归约为首范式 (head normal form)，关于首范式我们将在后面讨论。
- *byName* 用传名调用来规范项。

```
structure Reduce : REDUCE =
struct

  fun eval (Lambda.Apply(t1,t2)) =
    (case eval t1 of
      Lambda.Abs(a,u) => eval(Lambda.subst 0 (eval t2) u)
    | u1               => Lambda.Apply(u1, eval t2))
  | eval t              = t;

  fun byValue t          = bodies (eval t)
  and bodies (Lambda.Abs(a,t)) = Lambda.Abs(a, byValue t)
    | bodies (Lambda.Apply(t1,t2)) = Lambda.Apply(bodies t1, bodies t2)
    | bodies t                      = t;

  fun headNF (Lambda.Abs(a,t)) = Lambda.Abs(a, headNF t)
    | headNF (Lambda.Apply(t1,t2)) =
      (case headNF t1 of
        Lambda.Abs(a,t) => headNF(Lambda.subst 0 t2 t)
      | u1               => Lambda.Apply(u1, t2))
    | headNF t              = t;

  fun byName t          = args (headNF t)
  and args (Lambda.Abs(a,t)) = Lambda.Abs(a, args t)
    | args (Lambda.Apply(t1,t2)) = Lambda.Apply(args t1, byName t2)
    | args t                      = t;

end;
```

图9-8 λ-项的归约

传值调用。在ML中，对抽象 $\text{fn } x \Rightarrow E$ 求值并不会导致对 E 求值，因为在 x 值未知的情况下，通常没有方法来对 E 求值。我们经常利用ML对抽象的处理，通过书写 $\text{fn } () \Rightarrow E$ 来延迟对 E 的求值。这允许一定程度的惰性求值。

λ-演算的情况是不同的。抽象 $\lambda x.(\lambda y.a \ y) \ x$ 可以归约为范式 $\lambda x.a \ x$ ，而不需要理会 x 是否有值。即便如此，不去简化抽象的函数体是有好处的。因为这允许对求值延时，就像在ML里一样。这是处理递归所必需的。

函数 *eval*，给定应用 $t_1 \ t_2$ ，对 t_1 求值得到 u_1 ，对 t_2 求值得到 u_2 。（假设这些求值过程可以终止。）

如果 u_1 是抽象 $\lambda x.u$, 那么 $eval$ 将以 $u[u_2/x]$ 为参数调用自身, 来将参数值替换进函数体; 如果 u_1 是别的什么, 则 $eval$ 返回 $u_1 u_2$ 。若给定一个抽象或者变量, $eval$ 将原封不动地返回它的参数。虽然 $eval$ 进行了归约的大部分工作, 它的结果可能包含不是范式的抽象。

$byValue$ 借助 $eval$ 来将项归约为范式。它对其参数调用 $eval$, 然后递归地扫描结果, 以便规范其中的抽象。

假设 t 等于 $true$ 。当 $eval$ 的参数是 $if\ t\ u_1\ u_2$ 时, 它将对 u_1 和 u_2 两者都求值, 尽管只有 u_1 是必需的。如果这是一个递归函数的函数体, 那么它将永远运行下去, 就像在2.12节中所讨论的那样。我们应该插入抽象来延迟求值。选择任意的变量 x , 并将条件表达式编写为

$$(if\ t\ (\lambda x.u_1)\ (\lambda x.u_2))\ x$$

给定这一项, $eval$ 将返回 $\lambda x.u_1$ 作为 if 的结果, 并将它应用于 x 。因此它只会对 u_1 求值, 而不会对 u_2 求值。如果 t 等于 $false$, 则只会对 u_2 求值。在传值调用下, 条件表达式只能以这种方式编写。

用 Y 编写的递归定义在传值调用下会失败, 因为对 $Y\ f$ 的求值永远也停不下来。可以插入抽象到 Y 中去, 以便延迟求值。算子

$$YV \equiv \lambda f.(\lambda x.f(\lambda y.x\ x\ y))(\lambda x.f(\lambda y.x\ x\ y))$$

也具有不动点性质, 并且可以表示用 $byValue$ 求值的递归函数。

传名调用。一个 λ -项是首范式, 需满足对于 $m \geq 0$ 和 $n \geq 0$, 有如下形式:

$$\lambda x_1 \dots x_m.x\ t_1 \dots t_n$$

变量 x 可以是自由的也可以是被约束的(x_1, \dots, x_m 之一)。

可以看出该项的范式(如果存在)必定是

$$\lambda x_1 \dots x_m.x\ u_1 \dots u_n$$

其中 u_i 是 t_i 的范式($i = 1, \dots, n$)。首范式描述了项的外在结构, 它不会受归约影响。我们可以首先计算项的首范式, 然后递归地规范子项 t_1, \dots, t_n , 以达到规范整项的目的。如果存在范式, 这一过程最终会得到范式, 因为所有具有范式的项也具有首范式。

例如, 项 $\lambda x.a\ ((\lambda z.z)\ x)$ 是首范式, 而它的范式是 $\lambda x.a\ x$ 。不是首范式的项可以被看成

$$\lambda x_1 \dots x_m.(\lambda x.t)\ t_1 \dots t_n$$

其中 $n > 0$ 。它允许在函数体的最左测进行一次归约。例如, 对于所有项 t , $(\lambda x\ y.y\ x)\ t$ 都规约为首范式 $\lambda y.y\ t$ 。很多没有范式的项却有首范式, 例如

$$Y = \lambda f.f(Y\ f)$$

个别项, 例如 $(\lambda x.x\ x)\ (\lambda x.x\ x)$, 甚至没有首范式。这样的项可以被认为是无定义的。

函数 $headNF$ 通过递归地计算 $headNF\ t_1$ 来计算项 $t_1\ t_2$ 的首范式, 之后, 若结果是一个抽象就进行一个 β -转换。参数 t_2 在替换之前不会进行归约, 这是传名调用。^①

函数 $byName$ 是通过计算项的 $headNF$, 然后规范最外层应用的参数来规范项的。它以适当的效率完成了传名调用归约。

① $headNF$ 利用了Barendregt (1984) 中的命题8.3.13: 如果 $t\ u$ 是首范式, 那么 t 也是首范式。

练习 9.26 证明 $YVf = f(\lambda y.YVfy)$ 。

练习 9.27 推导出 Y 的一个首范式，或者说明不存在首范式。

练习 9.28 推导出 $inflist$ 的一个首范式，或者说明不存在首范式。

练习 9.29 推导出 $byValue$ 和 $byName$ 是怎样计算 $fst(pair\ t\ u)$ 的范式， t 和 u 为任意 λ -项。

9.13 演示求值程序

为了演示 λ -演算的实现，我们建立了环境 $stdEnv$ 。它定义了布尔值、序偶等在 λ -演算中的编码（图9-9）。结构 $StringDict$ 的函数 $insert$ 要将一个定义加入到字典中的条件是該字符串尚未在其中定义。

```
fun insertEnv ((a,b),env) =
  StringDict.insert (env, a, ParseTerm.read b);

val stdEnv = foldl insertEnv StringDict.empty
[
  (* 布尔 *)
  ("true", "%x y.x"),          ("false", "%x y.y"),
  ("if", "%p x y. p x y"),
  (* 序偶 *)
  ("pair", "%x y f.f x y"),
  ("fst", "%p.p true"),        ("snd", "%p.p false"),
  (* 自然数 *)
  ("suc", "%n f x. n f (f x)"),
  ("iszero", "%n. n (%x.false) true"),
  ("0", "%f x. x"),            ("1", "suc 0"),
  ("2", "suc 1"),              ("3", "suc 2"),
  ("4", "suc 3"),              ("5", "suc 4"),
  ("6", "suc 5"),              ("7", "suc 6"),
  ("8", "suc 7"),              ("9", "suc 8"),
  ("add", "%m n f x. m f (n f x)"),
  ("mult", "%m n f. m (n f)"),
  ("expt", "%m n f x. n m f x"),
  ("prefn", "%f p. pair (f (fst p)) (fst p)"),
  ("pre", "%n f x. snd (n (prefn f) (pair x x))"),
  ("sub", "%m n. n pre m"),
  (* 表 *)
  ("nil", "%z.z"),
  ("cons", "%x y. pair false (pair x y)"),
  ("null", "fst"),
  ("hd", "%z. fst(snd z)",      ("tl", "%z. snd(snd z)"),
  (* 传名调用的递归 *)
  ("Y", "%f. (%x.f(x x))(%x.f(x x))"),
  ("fact", "Y(%g n. if (iszero n) 1 (mult n (g (pre n))))"),
  ("append", "Y(%g z w. if (null z) w (cons (hd z) (g(tl z)w)))"),
  ("inflist", "Y(%z. cons MORE z)"),
  (* 传值调用的递归 *)
  ("YV", "%f. (%x.f(%y.x x y)) (%x.f(%y.x x y))"),
  ("factV",
    "YV (%g n. (if (iszero n) (%y.1) (%y.mult n (g (pre n))))y)"
  ];
```

图9-9 构造标准环境

函数`stdRead`读入一项，并根据`stdEnv`进行实例化，展开其中的定义。注意，“2”将展开成很大的项，是从`suc (suc 0)`推导出来的：

```
fun stdRead a = Lambda.inst stdEnv (ParseTerm.read a);
> val stdRead = fn : string -> Lambda.t
DisplayTerm.pr (stdRead "2");
> (%n f x. n f (f x))
> ((%n f x. n f (f x)) (%f x. x))
```

这一项可以进行规范。我们定义函数`try`使得`try evfn`读入一项，对其应用`evfn`，并显示结果。通过传值调用，我们将“2”规约为一个丘奇数：

```
fun try evfn = DisplayTerm.pr o evfn o stdRead;
> val try = fn : (lambda.t->lambda.t) -> string -> unit
try Reduce.byValue "2";
> %f x. f (f x)
```

传值调用可以进行简单的丘奇数算术运算： $2 + 3 = 5$ ， $2 \times 3 = 6$ ， $2^3 = 8$ ：

```
try Reduce.byValue "add 2 3";
> %f x. f (f (f (f (f x))))
try Reduce.byValue "mult 2 3";
> %f x. f (f (f (f (f (f x)))))
try Reduce.byValue "expt 2 3";
> %f x. f (f (f (f (f (f (f (f x)))))))
```

环境中定义了`factV`，它是用YV编码的递归阶乘函数，并使用抽象来延迟对`if`参数的求值。它可以在传值调用归约下执行，计算出 $3! = 6$ ：

```
try Reduce.byValue "factV 3";
> %f x. f (f (f (f (f (f x)))))
```

传名调用归约除可以完成传值调用能完成的计算外，还能完成更多的计算。它能处理涉及`Y`和`if`的递归定义了，不需使用任何技巧来延迟求值。下面，我们连接 (`append`) 表 [`FARE`, `THEE`]和 [`WELL`]:

```
try Reduce.byName
  "append (cons FARE (cons THEE nil)) (cons WELL nil)";
> %f. f (%x y. y)
> (%f. f FARE
> (%f. f (%x y. y)
> (%f. f THEE
> (%f. f (%x y. y)
> (%f. f WELL (%z. z))))))
```

让我们提取无穷表 [`MORE`, `MORE`, ...] 的首元素：

```
try Reduce.byName "hd inflist";
> MORE
```

执行极其缓慢，特别是使用传名调用时。计算 `fact 3` 需要330毫秒，相比之下`factV 3`只需60毫秒。计算`fact 4`需要40秒！这一点儿也不奇怪，因为算术使用的是一元记法，递归又是通过复制来进行的。尽管这样，我们也已经具备了函数式程序设计的所有元素。

再加一点努力我们就可以得到真正的函数式语言了。取代在纯λ-演算中编写数据结构的做法，我们可以将数、算术操作和序偶作为原语。为了在一个抽象机器上执行λ-项，我们可以对其进行编译而不是进行解释。对于传值调用归约来说，SECD机器是适合的。对于传名调用归约，我们可以将λ-项编译为组合子，并通过图归约来执行。设计和实现一个简单的函数式语言是一个有挑战性的项目。

i 进一步的阅读。M. J. C. Gordon (1988) 从计算机科学家的角度讲述了λ-演算，他讨论了数据的表示方法，并给出了归约和转换λ-表达式的Lisp代码。Barendregt (1984) 是λ-演算的一个全面的参考。Boolos和Jeffrey (1980) 介绍了可计算性理论，包括图灵机、寄存器机器和广义递归函数。

N. G. de Bruijn (1972) 开发了λ-演算的无名记法，并用于他的AUTOMATH系统 (Nederpelt等, 1994)。它也被用于Isabelle (Paulson, 1994) 和第10章的定理证明机Hal。

Field和Harrison (1988) 讲述了基本的组合子归约。新式的惰性求值实现使用了更为成熟的技术 (Peyton Jones, 1992)。

393
395

练习 9.30 当应用try *Reduce.byName*到下面的字符串上时将得到什么结果?

```
"hd (tl (Y (%z. append (cons MORE (cons AND nil)) z)))"
"hd (tl (tl (Y (%g n. cons n (g (suc n))) 0)))"
```

要点小结

- 自顶向下的语法分析可以通过高阶函数自然地表示。
- λ-演算是一个计算的理论模型，它与函数式程序设计非常类似。
- 变量约束的无名表示法易于在计算机上实现。
- 数和表这样的数据结构，以及它们的操作都可以编写为λ-项。
- λ-项Y通过重复复制来对递归进行编码。
- 对于λ-演算来说，存在传值调用求值策略和传名调用求值策略。

396

第10章 策略定理证明机

ML原先是为实现定理证明机（爱丁堡LCF）而设计的程序设计语言。因此一本关于ML的书以讲述一个定理证明机而结束是合适的。这个定理证明机称为Hal，源自LCF。[⊖]Hal通过从证明的目标开始反向逐步求精来构造证明。从最简单的方面看，这是证明检测：每一步，都有一条推理规则和目标进行匹配，将其简化成某些子目标。如果要证明一些有意义的东西，更多的方面需要自动化。Hal提供策略（tactic）和策略算子（tactical），它们构成了一种表示搜索过程的高级语言。几个基本策略通过深度优先搜索的策略算子加以应用，可以实现一个一般性的策略，该策略能够自动证明许多定理，例如：

$$\begin{aligned}\neg(\exists x. \forall y. \phi(x, y) \leftrightarrow \neg\phi(y, y)) \\ \exists xy. \phi(x, y) \rightarrow \forall xy. \phi(x, y) \\ \exists x. \forall yz. (\phi(y) \rightarrow \psi(z)) \rightarrow (\phi(x) \rightarrow \psi(x))\end{aligned}$$

从纯粹的能力上讲，Hal不能和专门的定理证明机相比。Hal牺牲了能力来换取一些灵活性。一个典型的归结定理证明机支持带相等的纯古典逻辑，但是没有归纳。策略定理证明机允许在几乎任意逻辑中使用自动和交互式的混合工作方式。

Hal可用于古典逻辑是由于它广为人知，Hal也可以容易地扩展到推理、模态算子、集合论或其他方面。需要改变它的策略以反映新的推理规则，策略算子保持不变，随时可以表示新增逻辑的搜索过程。

本章提要

本章包含以下几节：

- 一阶逻辑的相继式演算。这里简要勾勒了一阶逻辑的语义，并讲述了相继式演算。量词推理涉及了参数和元变量。
- 在ML中处理项和公式。一阶逻辑的Hal表示借鉴了前面章节的技术。一项主要的新技术是合一。
- 策略和证明状态。Hal将相继式演算实现为一套在证明状态的抽象类型上的变换。每条推理规则都作为一个策略。
- 搜索证明。用户界面比较粗糙，不过可以演示策略。策略算子为策略增添了控制结构，并被用来给一阶逻辑编写自动策略。

397

一阶逻辑的相继式演算

我们从一阶逻辑的简单介绍开始。一阶逻辑的语法已经在6.1节给出。命题逻辑

⊖ Hal是以亨利五世王命名的，他是一个出色的战术家。

(propositional logic) 只关心由连接词 \wedge 、 \vee 、 \neg 、 \rightarrow 和 \leftrightarrow 构成的公式。一阶逻辑引入了量词 \forall 和 \exists ，以及变量和项。一阶语言 (first-order language) 在逻辑符号的基础上补充了某些常量 a, b, \dots 、函数符号 f, g, \dots 和谓词符号 P, Q, \dots 。令 ϕ, ψ, χ, \dots 代表任意公式。

全域 (universe) 是一个非空集合，它包含了项的所有可能值。常量表示全域中的元素，函数符号表示全域中的函数，谓词符号表示全域中的关系。结构 (structure) 定义了一阶语言的语义，它描述了一个全域，并给出了常量、函数符号和谓词符号的解释。ML 结构类似于逻辑结构。

公式的含义取决于公式中自由变量的值。赋值 (assignment) 是从自由变量到全域元素的一个映射。给定一个结构和一个赋值，每个公式或为真或为假。公式 $\forall x. \phi$ 为真当且仅当对于所有可能赋给 x 的值 ϕ 都为真 (保持其他变量不变)。连接词由真值表定义，例如， $\phi \wedge \psi$ 为真当且仅当 ϕ 为真且 ψ 为真。

有效的公式是指在所有结构和赋值下都为真的公式。由于存在无穷多个结构，因此不能用穷尽测试去证明一个公式是有效的。我们可以通过基于推理规则的形式证明来证明公式的有效性，推理规则的正确性是由逻辑语义保证的。每条规则接受零个或多个前提并产生一个结论，一条合理的规则在前提有效的情况下会产生有效的结论。一套关于逻辑的推理规则称为证明系统或形式化。

398

在众多古典一阶逻辑的证明系统中，最容易自动化的是相继式演算 (sequent calculus)。表方法有时用于自动化一阶逻辑，是相继式演算的一种紧凑记法。

10.1 命题逻辑的相继式演算

为了简单起见，让我们暂时将注意力放在命题逻辑上。一个相继式具有如下形式

$$\phi_1, \dots, \phi_m \vdash \psi_1, \dots, \psi_n$$

其中 ϕ_1, \dots, ϕ_m 和 ψ_1, \dots, ψ_n 是公式的多重集合。如第6章所述，多重集合是一个顺序无关紧要的元素集。传统上相继式包含的是公式表，而逻辑中有一些规则是用于交换表中相邻公式的，多重集合可以省去这些规则。

给定一个结构和赋值，上面的相继式为真当且仅当公式 ϕ_1, \dots, ϕ_m 中有些为假，或公式 ψ_1, \dots, ψ_n 中有些为真。换句话说，这个相继式和下面的公式含义相同

$$\phi_1 \wedge \dots \wedge \phi_m \rightarrow \psi_1 \vee \dots \vee \psi_n$$

作为一个特别情形， $\vdash \psi$ 的含义和 ψ 相同，不过 \vdash 符号 (绕杆符) 不是一个逻辑连接词。

为了方便书写规则， Γ 和 Δ 用来代表公式的多重集合。逗号表示多重集合的并，因此 Γ, Δ 代表 Γ 和 Δ 的并集。在需要多重集合的地方出现的公式 (像 $\Gamma \vdash \phi$ 中的 ϕ) 表示了一个元素的多重集合。因此， Γ, ϕ 是包含至少一个 ϕ 的多重集合，其中 Γ 表示多重集合中的其他元素。

有效性和基本相继式。有效相继式是指在所有结构和赋值下都为真的相继式。相继式演算中的定理就是那些有效相继式。

当两边都含有一个公共的公式 ϕ 时，相继式是基本的 (basic)。这可以形式化地写为公理

$$\phi, \Gamma \vdash \Delta, \phi$$

根据刚才讲述的记法, ϕ, Γ 和 Δ, ϕ 都是含有 ϕ 的多重集合。这样的相继式显然是有效的。

那些包含于 Γ 和 Δ 中的其他公式在推理中不起作用。有时要对相继式演算进行整理以使基本相继式具有形式 $\phi \vdash \phi$ 。然后, 形如 $\phi, \Gamma \vdash \Delta, \phi$ 的相继式就可以借助“弱化”规则推导出来了, 该规则可以将任意公式插入相继式中。

连接词的相继式规则。相继式演算的规则是成对出现的, 分别将每个连接词插入到 \vdash 符号的左侧或右侧。例如, 规则 $\wedge:\text{left}$ 将合取插入左侧, 而 $\wedge:\text{right}$ 则将合取插入右侧。下面是后者的常用记法, 前提在上, 结论在下:

$$\frac{\Gamma \vdash \Delta, \phi \quad \Gamma \vdash \Delta, \psi}{\Gamma \vdash \Delta, \phi \wedge \psi} \wedge:\text{right}$$

为了证明 $\wedge:\text{right}$ 是一条合理的规则, 我们假设它的前提是有效的并说明结论也是有效的。假设在某些结构和赋值下, Γ 中的所有公式都为真, 我们必须说明 $\Delta, \phi \wedge \psi$ 中的某个公式也为真。如果 Δ 中没有公式为真, 那么根据前提 ϕ 和 ψ 两者都为真。因此 $\phi \wedge \psi$ 为真。

现在来说明规则 $\wedge:\text{left}$ 的正确性。

$$\frac{\phi, \psi, \Gamma \vdash \Delta}{\phi \wedge \psi, \Gamma \vdash \Delta} \wedge:\text{left}$$

为了说明这条规则合理, 我们像上面那样进行证明。假设 $\Gamma, \phi \wedge \psi$ 中的所有公式为真, 那么 ϕ 和 ψ 两者都为真。在前提有效的假设下, Δ 中的某个公式必须为真, 也就得到了结论。

图10-1给出了命题连接词 \wedge 、 \vee 、 \rightarrow 、 \leftrightarrow 和 \neg 的规则。所有规则的论证都是类似的。

$:\text{left}$	$:\text{right}$
$\frac{\phi, \psi, \Gamma \vdash \Delta}{\phi \wedge \psi, \Gamma \vdash \Delta}$	$\frac{\Gamma \vdash \Delta, \phi \quad \Gamma \vdash \Delta, \psi}{\Gamma \vdash \Delta, \phi \wedge \psi}$
$\frac{\phi, \Gamma \vdash \Delta \quad \psi, \Gamma \vdash \Delta}{\phi \vee \psi, \Gamma \vdash \Delta}$	$\frac{\Gamma \vdash \Delta, \phi, \psi}{\Gamma \vdash \Delta, \phi \vee \psi}$
$\frac{\Gamma \vdash \Delta, \phi \quad \psi, \Gamma \vdash \Delta}{\phi \rightarrow \psi, \Gamma \vdash \Delta}$	$\frac{\phi, \Gamma \vdash \Delta, \psi}{\Gamma \vdash \Delta, \phi \rightarrow \psi}$
$\frac{\phi, \psi, \Gamma \vdash \Delta \quad \Gamma \vdash \Delta, \phi, \psi}{\phi \leftrightarrow \psi, \Gamma \vdash \Delta}$	$\frac{\phi, \Gamma \vdash \Delta, \psi \quad \psi, \Gamma \vdash \Delta, \phi}{\Gamma \vdash \Delta, \phi \leftrightarrow \psi}$
$\frac{\Gamma \vdash \Delta, \phi}{\neg \phi, \Gamma \vdash \Delta}$	$\frac{\phi, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg \phi}$

图10-1 命题连接词的相继式规则

练习 10.1 哪一个公式和右侧为空的规则 $\phi_1, \dots, \phi_m \vdash$ 等价?

练习 10.2 证明规则 $\vee:\text{left}$ 和 $\vee:\text{right}$ 是正确的。

练习 10.3 证明规则 $\leftrightarrow:\text{left}$ 和 $\leftrightarrow:\text{right}$ 是正确的。

10.2 证明相继式演算中的定理

推理规则通常是顺正向 (由上至下) 阅读的, 从前提到结论。因此 $\wedge:\text{right}$ 接受前提 $\Gamma \vdash$

Δ, ϕ 和 $\Gamma \vdash \Delta, \psi$, 产生结论 $\Gamma \vdash \Delta, \phi \wedge \psi$ 。对结论的相继式应用另一个规则将产生另一个结论, 依此类推。一个形式证明是通过应用推理规则构造的一棵树。下面是相继式 $\phi \wedge \psi \vdash \psi \wedge \phi$ 的证明:

$$\frac{\frac{\phi, \psi \vdash \psi \quad \phi, \psi \vdash \phi}{\phi, \psi \vdash \psi \wedge \phi} \wedge:\text{right}}{\phi \wedge \psi \vdash \psi \wedge \phi} \wedge:\text{left} \quad (*)$$

从正向看, 首先两个基本相继式通过 $\wedge:\text{right}$ 组合, 然后其结论再经 $\wedge:\text{left}$ 变换。但是, 正向阅读并不能帮助我们找到一个给定相继式的证明。

为了达到发现证明的目的, 应从反向(由下至上)去看规则, 从目标到子目标。因此, $\wedge:\text{right}$ 接受目标 $\Gamma \vdash \Delta, \phi \wedge \psi$ 并返回子目标 $\Gamma \vdash \Delta, \phi$ 和 $\Gamma \vdash \Delta, \psi$ 。如果可以证明这些子目标是定理, 那么该目标也是。子目标将通过进一步应用规则来求精, 直到剩下的所有子目标都是基本相继式为止, 这时立即可知它们是有用的。证明树是由根向上构造的, 这一过程被称为求精(refinement)或反向证明(backward proof)。

从反向看, 证明(*)从要证明的相继式开始, 也就是 $\phi \wedge \psi \vdash \psi \wedge \phi$ 。这一目标通过 $\wedge:\text{left}$ 被求精到 $\phi, \psi \vdash \psi \wedge \phi$, 这一子目标通过 $\wedge:\text{right}$ 被求精到 $\phi, \psi \vdash \psi$ 和 $\phi, \psi \vdash \phi$ 。最后这两个子目标都是基本相继式, 证明也就结束了。

在反向阅读下, 每条规则拆解目标中的一个公式。应用 $\wedge:\text{left}$ 将左侧的合取拆开, 应用 $\wedge:\text{right}$ 将右侧的合取拆开。如果所有结果子目标都是基本相继式, 那么初始目标就得证了。对于命题逻辑, 这一过程一定会终止。

一个相继式可以有多个不同的证明, 这取决于首先拆开哪个公式。证明(*)首先拆开了 $\phi \wedge \psi \vdash \psi \wedge \phi$ 左侧的合取式。为了得到一个不同的证明, 可以首先拆开右侧的合取式:

$$\frac{\frac{\phi, \psi \vdash \psi}{\phi \wedge \psi \vdash \psi} \wedge:\text{left} \quad \frac{\phi, \psi \vdash \phi}{\phi \wedge \psi \vdash \phi} \wedge:\text{left}}{\phi \wedge \psi \vdash \psi \wedge \phi} \wedge:\text{right}$$

这比证明(*)要长, 因为 $\wedge:\text{left}$ 被应用了两次。对初始目标应用 $\wedge:\text{right}$ 产生了两个子目标, 每个的左侧都有一个合取式。如果每一步都选择产生最少子目标的规则, 则通常会得到较短的证明。

总结我们的证明步骤如下:

- 取得要证明的相继式作为初始目标。证明树的根, 也是唯一的叶子, 就是这个目标。
- 选取某个证明树的叶子作为子目标, 并对其应用一条规则, 将这一叶子转变为一个具有一个或多个叶子的分支结点。
- 在所有叶子都是基本相继式时停止(证明成功), 或在找不到可以应用到一个叶子(子目标)上的规则时停止(证明失败)。

这套步骤非常有效, 尽管它的搜索是不确定的。 $\vee:\text{left}$ 和 $\wedge:\text{right}$ 都可以应用于子目标 $p \vee q, r \vdash r \wedge r$ 。前一规则对无关的 $p \vee q$ 进行情形分析; 后一规则产生两个基本的子目标, 证明立即成功。

练习 10.4 构造相继式 $\phi \vee \psi \vdash \psi \vee \phi$ 和 $\phi_1 \wedge (\phi_2 \wedge \phi_3) \vdash (\phi_1 \wedge \phi_2) \wedge \phi_3$ 的证明。

练习 10.5 构造下面相继式的证明

$$\vdash (\phi_1 \wedge \phi_2) \vee \psi \leftrightarrow (\phi_1 \vee \psi) \wedge (\phi_2 \vee \psi)$$

400
402练习 10.6 证明任何 \vdash 符号左侧同时含有 ϕ 和 $\neg \phi$ 的相继式是有效的。

10.3 量词的相继式规则

命题逻辑是可判定的，我们的证明步骤在有限的时间内就可以确定任意一个公式是否为定理。当加入量词时，就不存在这样的判定步骤了。此外，量词带来了许多语法上的复杂问题。

每个量词约束一个变量，因此在 $\forall x. \exists y. R(x, y, z)$ 中 x 和 y 是被约束的，而 z 是自由的。重新命名约束变量不会影响公式的含义，前例和 $\forall y. \exists w. R(y, w, z)$ 是等价的。有些推理规则涉及了替换， $\phi[t/x]$ 则代表了在 ϕ 中将所有自由的 x 都替换成 t 所产生的结果。不那么正式的话， $\phi(x)$ 代表了含有 x 的公式，而 $\phi(t)$ 代表了将自由的 x 替换成 t 以后的结果。 λ -演算语法中约束变量的无名表示法（9.6节）同样适用于量词语法。

全称量词有下面两条相继式规则：

$$\frac{\phi[t/x], \forall x. \phi, \Gamma \vdash \Delta}{\forall x. \phi, \Gamma \vdash \Delta} \forall:\text{left} \quad \frac{\Gamma \vdash \Delta, \phi}{\Gamma \vdash \Delta, \forall x. \phi} \forall:\text{right}$$

限制条件： x 不能在结论中自由出现

规则 $\forall:\text{left}$ 容易理解，如果 $\forall x. \phi$ 为真，那么 $\phi[t/x]$ 也为真，其中 t 是任意项。

为了证明更为复杂的规则 $\forall:\text{right}$ 是正确的，我们假设它的前提有效，然后证明其结论也有效。给定某个结构和赋值，假设所有 Γ 中的公式都为真，并且 Δ 中没有公式为真，那么我们必须证明 $\forall x. \phi$ 为真。这只要证明对于每一个可能的 x 赋值来说，在其他变量保持不变的情况下 ϕ 为真。根据 $\forall:\text{right}$ 的限制条件，改变 x 的值不会影响 Γ 和 Δ 中任何公式的真值，又由于前提有效，则 ϕ 一定为真。

忽略这个限制条件可能会产生荒谬的推理：

$$\frac{P(x) \vdash P(x)}{P(x) \vdash \forall x. P(x)} \forall:\text{right}$$

当 $P(x)$ 代表整数上的谓词 $x = 0$ ，并且 x 被赋值为 0 时，这个结论为假。

存在量词有下面两条相继式规则：

$$\frac{\phi, \Gamma \vdash \Delta}{\exists x. \phi, \Gamma \vdash \Delta} \exists:\text{left} \quad \frac{\Gamma \vdash \Delta, \exists x. \phi, \phi[t/x]}{\Gamma \vdash \Delta, \exists x. \phi} \exists:\text{right}$$

限制条件： x 不能在结论中自由出现

它们是全称量词规则的对偶，可以类似地进行论证。注意， $\exists x. \phi$ 等价于 $\neg \forall x. \neg \phi$ 。

规则 $\forall:\text{left}$ 和 $\exists:\text{right}$ 有一个其他规则所没有的特点：在反向证明中，它们不从目标中删除任何公式。它们展开一个量词公式，把一项替换进公式体内，同时保留这个公式并允许重复展开。事先确定证明中需要对一个量词公式进行多少次展开是不可能的。正因为如此，我们的证明步骤可能停不下来，一阶逻辑是不可判定的。

练习 10.7 如果忽略 $\forall:\text{right}$ 的限制条件，用到这一规则的证明会得到不一致的结论吗？（这意味着有一个相继式 $\vdash \phi$ 使得 $\neg \phi$ 是个有效公式。）

403

10.4 带量词的定理证明

我们的反向证明步骤对于量词相当有效，至少在对付简单问题上不需要更具识别能力的搜索。我们从一个涉及全称量词的简单证明开始：

$$\begin{array}{c}
 \frac{\phi(x), \forall x. \phi(x) \vdash \phi(x), \psi(x)}{\forall x. \phi(x) \vdash \phi(x), \psi(x)} \quad \forall:\text{left} \\
 \frac{\forall x. \phi(x) \vdash \phi(x), \psi(x)}{\forall x. \phi(x) \vdash \phi(x) \vee \psi(x)} \quad \forall:\text{right} \\
 \frac{}{\forall x. \phi(x) \vdash \forall x. \phi(x) \vee \psi(x)} \quad \forall:\text{right}
 \end{array}$$

$\forall:\text{right}$ 的限制条件成立， x 在结论中不是自由的。在反向证明中，这一结论是初始目标。

如果我们首先应用 $\forall:\text{left}$ 来插入公式 $\phi(x)$ ，那么 x 在子目标中就是自由的了。之后则必须重新命名量词变量才能应用 $\forall:\text{right}$ ：

$$\begin{array}{c}
 \frac{\phi(x), \forall x. \phi(x) \vdash \phi(y), \psi(y)}{\phi(x), \forall x. \phi(x) \vdash \phi(y) \vee \psi(y)} \quad \forall:\text{right} \\
 \frac{\phi(x), \forall x. \phi(x) \vdash \phi(y) \vee \psi(y)}{\phi(x), \forall x. \phi(x) \vdash \forall x. \phi(x) \vee \psi(x)} \quad \forall:\text{right} \\
 \frac{}{\forall x. \phi(x) \vdash \forall x. \phi(x) \vee \psi(x)} \quad \forall:\text{left}
 \end{array}$$

最上面的相继式并不是基本的，要完成证明必须再次应用 $\forall:\text{left}$ ，第一次应用这条规则没起到什么作用。我们有了一个一般性的启发式原则：如果有另一规则可以有效地应用到目标上，则不要去使用 $\forall:\text{left}$ 和 $\exists:\text{right}$ 。

404

下面的证明说明了使用量词的一些困难。[⊖]

$$\begin{array}{c}
 \frac{\phi(x), \phi(z) \vdash \exists z. \phi(z) \rightarrow \forall x. \phi(x), \phi(x), \forall x. \phi(x)}{\phi(z) \vdash \exists z. \phi(z) \rightarrow \forall x. \phi(x), \phi(x), \phi(x) \rightarrow \forall x. \phi(x)} \quad \rightarrow:\text{right} \\
 \frac{\phi(z) \vdash \exists z. \phi(z) \rightarrow \forall x. \phi(x), \phi(x)}{\phi(z) \vdash \exists z. \phi(z) \rightarrow \forall x. \phi(x), \forall x. \phi(x)} \quad \exists:\text{right} \\
 \frac{\phi(z) \vdash \exists z. \phi(z) \rightarrow \forall x. \phi(x), \forall x. \phi(x)}{\vdash \exists z. \phi(z) \rightarrow \forall x. \phi(x), \phi(z) \rightarrow \forall x. \phi(x)} \quad \forall:\text{right} \\
 \frac{}{\vdash \exists z. \phi(z) \rightarrow \forall x. \phi(x)} \quad \rightarrow:\text{right}
 \end{array}$$

从目标向上进行，应用 $\exists:\text{right}$ 加进了 z 作为自由变量。虽然存在量词公式仍在子目标中，但是它将保持静止直到再次到达一个没有其他规则可以应用的目标。下一步推理 $\rightarrow:\text{right}$ 将 $\phi(z)$ 移到左侧。由于 x 在子目标中不是自由的，因此可以应用 $\forall:\text{right}$ ，将 $\forall x. \phi(x)$ 换成了 $\phi(x)$ 。在结果子目标中再次应用了 $\exists:\text{right}$ （没有其他选择），用 x 替换 z 。在 $\rightarrow:\text{right}$ 之后，最终的子目标是一个基本相继式，它在两边都有 $\phi(x)$ 。

注意 $\exists z. \phi(z) \rightarrow \forall x. \phi(x)$ 通过 $\exists:\text{right}$ 被展开了两次。这个相继式没有其他证明方法。对于任意给定的 n ，都不难设计出需要展开 n 次的相继式。

合一。在量词的论证中，有一个难点：在规则 $\exists:\text{right}$ 和 $\forall:\text{left}$ 中如何选择项 t 。这相当于预测哪一项会最终产生基本子目标和一个成功的证明。在上面的证明中，第一次 $\exists:\text{right}$ 时对 z 的选择是任意的，任何项都能行。第二次 $\exists:\text{right}$ 时选择 x 是关键——但可能并不明显。

我们可以在这样的规则中推迟项的选择。我们引入元变量（meta-variable） $?a, ?b, \dots$ 作为占位项。当一个目标可以通过将它的元变量替换成适当的项来解决时，就在整个证明中进行这一替换。例如，对于子目标 $P(?a), \Gamma \vdash \Delta, P(f(?b))$ 来说，当我们将 $?a$ 换成 $f(?b)$ 时，它就成

⊖ 为了看出 $\exists z. \phi(z) \rightarrow \forall x. \phi(x)$ 是一条定理，首先注意它的全括号形式为： $\exists z. [\phi(z) \rightarrow (\forall x. \phi(x))]$ 。将存在量词推进蕴涵式内部则将其变为全称量词。于是这个公式就等价于 $(\forall z. \phi(z)) \rightarrow (\forall x. \phi(x))$ ，这显然为真。

为基本的了，可以看到， $?a$ 尚未完全确定，只是确定了它的外层形式 $f(\dots)$ 。未知项以递增的方式逐步解决。合一 (unification) 这一确定适当替换的过程，是量词论证的关键。

规则 $\forall:\text{left}$ 现在具有如下形式，其中 $?a$ 代表任意元变量：

$$\frac{\phi[?a/x], \forall x. \phi, \Gamma \vdash \Delta}{\forall x. \phi, \Gamma \vdash \Delta} \forall:\text{left}$$

强限制条件。元变量自身也有麻烦。回忆一下， $\forall:\text{right}$ 和 $\exists:\text{left}$ 具有限制条件 “ x 不能在结论中自由出现”。如果结论中含有元变量又怎么办呢？这些元变量可能被任何项替换掉。我们的方案是将每个自由变量标记上一个禁止元变量表。自由变量 $b_{?a_1, \dots, ?a_k}$ 不能被包含在用于替换元变量 $?a_1, \dots, ?a_k$ 的项内。合一算法可以强制这一点。

让我们简化一下术语。带标签的自由变量称为参数。元变量则称为变量。

使用参数，规则 $\forall:\text{right}$ 变为

$$\frac{\Gamma \vdash \Delta, \phi[b_{?a_1, \dots, ?a_k}/x]}{\Gamma \vdash \Delta, \forall x. \phi} \forall:\text{right} \quad \begin{array}{l} \text{限制条件: } b \text{ 不能出现在结论中,} \\ \text{且 } ?a_1, \dots, ?a_k \text{ 是结论中所有的变量.} \end{array}$$

限制条件的第一部分保证了 b 尚未被使用，而第二部分保证了 b 在稍后的替换中不会溜进来。对 $\exists:\text{left}$ 的处理也是一样的。

参数保证了正确的量词论证。例如， $\forall x. \phi(x, x)$ 一般来说并不蕴涵 $\exists y. \forall x. \phi(x, y)$ 。看看下面对相应相继式的尝试证明：

$$\begin{array}{rcl} \frac{\phi(?c, ?c), \forall x. \phi(x, x) \vdash \exists y. \forall x. \phi(x, y), \phi(b_{?a}, ?a)}{\forall x. \phi(x, x) \vdash \exists y. \forall x. \phi(x, y), \phi(b_{?a}, ?a)} & \forall:\text{left} \\ \frac{\forall x. \phi(x, x) \vdash \exists y. \forall x. \phi(x, y), \phi(b_{?a}, ?a)}{\forall x. \phi(x, x) \vdash \exists y. \forall x. \phi(x, y), \forall x. \phi(x, ?a)} & \forall:\text{right} \\ \frac{\forall x. \phi(x, x) \vdash \exists y. \forall x. \phi(x, y), \forall x. \phi(x, ?a)}{\forall x. \phi(x, x) \vdash \exists y. \forall x. \phi(x, y)} & \exists:\text{right} \end{array}$$

最上面的相继式不可能变成基本的。要想使 $\phi(?c, ?c)$ 和 $\phi(b_{?a}, ?a)$ 相等，必须有一个替换将 $?c$ 和 $?a$ 都换成 $b_{?a}$ 。然而，参数 $b_{?a}$ 被禁止出现在替换 $?a$ 的项中。这一尝试性证明会继续通过应用 $\forall:\text{right}$ 和 $\exists:\text{left}$ 向上增长，但永远不会产生基本相继式。

为了对比，我们来证明例子 $\forall x. \phi(x, x)$ 蕴涵 $\forall x. \exists y. \phi(x, y)$ ：

$$\begin{array}{rcl} \frac{\phi(?c, ?c), \forall x. \phi(x, x) \vdash \exists y. \phi(a, y), \phi(a, ?b)}{\forall x. \phi(x, x) \vdash \exists y. \phi(a, y), \phi(a, ?b)} & \forall:\text{left} \\ \frac{\forall x. \phi(x, x) \vdash \exists y. \phi(a, y), \phi(a, ?b)}{\forall x. \phi(x, x) \vdash \exists y. \phi(a, y)} & \exists:\text{right} \\ \frac{\forall x. \phi(x, x) \vdash \exists y. \phi(a, y)}{\forall x. \phi(x, x) \vdash \forall x. \exists y. \phi(x, y)} & \forall:\text{right} \end{array}$$

将 $?b$ 和 $?c$ 都替换成 a ，可以把 $\phi(?c, ?c)$ 和 $\phi(a, ?b)$ 都变换成 $\phi(a, a)$ ，证明就结束了。参数 a 没有标记任何变量，因为在供给 $\forall:\text{right}$ 的目标中没有变量。

❶ 进一步的阅读。不少教科书都是从计算机科学的角讲述逻辑的。它们把重点放在证明步骤与合一上，回避了数理逻辑所关心的更为传统的问题，例如模型论。关于逻辑的初步介绍，参见 Galton (1990) 或 Reeves 和 Clarke (1990)。Gallier (1986) 给出的是一个以相继式演算为中心的更具技术性的讲解。

练习 10.8 重新构造上面的前三个量词证明，这回使用 (元) 变量和参数。

练习 10.9 通过让 P 来表示某个结构里的一个恰当关系来举出相继式 $\forall x. P(x, x) \vdash \exists y. \forall x. P(x, y)$

的反例。

练习 10.10 如果对 $\forall x. \phi(x, x) \vdash \exists y. \forall x. \phi(x, y)$ 的尝试证明继续进行下去的话, 参数能不能使它成功?

练习 10.11 说明仅应用一次 \exists :right 无法证明 $\vdash \exists z. \phi(z) \rightarrow \forall x. \phi(x)$ 。

练习 10.12 对于下面每条构造一个证明, 或说明不存在证明 (a 和 b 是常量):

$$\begin{aligned} & \vdash \exists z. \phi(z) \rightarrow \phi(a) \wedge \phi(b) \\ & \forall x. \exists y. \phi(x, y) \vdash \exists y. \forall x. \phi(x, y) \\ & \exists y. \forall x. \phi(x, y) \vdash \forall x. \exists y. \phi(x, y) \end{aligned}$$

在ML中处理项和公式

我们来为定理证明编写一个框架。它必须能表示项和公式, 同时实现抽象、替换、语法分析和美化打印。要感谢前面章节积累起来的方法, 它们使得这项程序设计任务中没有什么特别困难的。

10.5 表示项和公式

我们为 λ -演算开发的技术(9.7节)也可用于一阶逻辑。在某些方面, 一阶逻辑更为简单。推理只能影响最外层的变量约束, 没有与 λ -项中的归约相对应的东西。

签名。签名FOL定义了一阶逻辑中项和公式的表示:

```
signature FOL =
sig
  datatype term = Var      of string
                | Param    of string * string list
                | Bound    of int
                | Fun      of string * term list
  datatype form = Pred     of string * term list
                | Conn     of string * form list
                | Quant    of string * string * form
  type goal      = form list * form list
  val precOf     : string -> int
  val abstract   : int -> term -> form -> form
  val subst      : int -> term -> form -> form
  val termVars   : term * string list -> string list
  val goalVars   : goal * string list -> string list
  val termParams : term * (string * string list) list
                  -> (string * string list) list
  val goalParams : goal * (string * string list) list
                  -> (string * string list) list
end;
```

类型term实现了上一节所讲的方法。变量(构造子Var)有一个名字。约束(Bound)变量有一个索引。函数(Fun)应用有函数名和参数表, 没有参数的函数只是一个常量。参数(Param)有一个名字和一个禁止变量表。

类型form是基本的。原子公式(Pred)有一个谓词的名字和参数表。连接词应用(Conn)

有一个连接词和一个公式表，通常都是" \sim "、" $\&$ "、" \mid "、" \rightarrow "或" \leftrightarrow "和一个或两个公式(的表)组成的序偶。 $Quant$ 公式有一个量词("ALL"或"EX"之一)，一个约束变量名字和一个公式作为公式体。

类型 $goal$ 是对公式表序偶类型的缩写。有些早期的ML编译器不允许在签名中使用类型缩写。我们可以简单地将 $goal$ 描述为一个类型：它在结构中的声名就会对外可见。

函数 $precOf$ 定义了连接词的优先级，这是语法分析和美化打印所必需的。

函数 $abstract$ 和 $subst$ 类似于上一章的同名函数。调用 $abstract\ i\ t\ p$ 把 p 中每一个 t 都替换成索引 i （它在各量词限定域中递增），典型情况是 $i = 0$ ， t 为一个原子项。调用 $subst\ i\ t\ p$ 把公式 p 中的索引 i （在量词限定域中递增）替换成 t 。

函数 $termVars$ 收集了项中的变量表（没有重复）， $termVars(t, bs)$ 将 t 中的所有变量插入到了表 bs 中。参数 bs 可能看上去有些复杂且没有必要，但是它省去了代价很高的表追加操作，同时允许 $termVars$ 被推广到公式和目标上去。这一点在看到函数的具体定义时会变得更清楚。

函数 $goalVars$ 也有两个参数，它收集目标的变量表。在Hal中，目标是一个相继式。虽然在ML中相继式是用公式表表示的，而不是用多重集合，但是我们应该可以实现上面讨论过的证明方式。

函数 $termParams$ 和 $goalParams$ 相应地收集了项或目标的参数表。每个参数都由它的名字和与之相关联的变量名表构成。

结构。结构 Fol （图10-2）实现了签名 FOL 。为了节省版面空间，其中 $term$ 和 $form$ 的数据类型（datatype）声明被略去了，它们和签名中的完全一样。该结构定义了几个签名中没有描述的函数。

```
structure Fol : FOL =
  struct
    datatype term = ...; datatype form = ...
    type goal = form list * form list;

    fun replace (u1,u2) t =
      if t=u1 then u2 else
        case t of Fun(a,ts) => Fun(a, map (replace (u1,u2)) ts)
          | _ => t;

    fun abstract i t (Pred(a,ts)) = Pred(a, map (replace(t, Bound i)) ts)
      | abstract i t (Conn(b,ps)) = Conn(b, map (abstract i t) ps)
      | abstract i t (Quant(qnt,b,p)) = Quant(qnt, b, abstract (i+1) t p);

    fun subst i t (Pred(a,ts)) = Pred(a, map (replace(Bound i, t)) ts)
      | subst i t (Conn(b,ps)) = Conn(b, map (subst i t) ps)
      | subst i t (Quant(qnt,b,p)) = Quant(qnt, b, subst (i+1) t p);

    fun precOf "~" = 4
      | precOf "&" = 3
      | precOf "|" = 2
      | precOf "<->" = 1
      | precOf "-->" = 1
      | precOf _ = ~1 (* 意味着不是一个中缀 *);
```

图10-2 一阶逻辑：表示项和公式

```

fun accumForm f (Pred(_,ts), z)    = foldr f z ts
  | accumForm f (Conn(_,ps), z)    = foldr (accumForm f) z ps
  | accumForm f (Quant(_,_,p), z) = accumForm f (p,z);

fun accumGoal f ((ps,qs), z) = foldr f (foldr f z qs) ps;

fun insert ...

fun termVars (Var a, bs)          = insert(a,bs)
  | termVars (Fun(_,ts), bs)     = foldr termVars bs ts
  | termVars (_, bs)             = bs;

val goalVars = accumGoal (accumForm termVars);

fun termParams (Param(a,bs), pairs) = (a,bs) :: pairs
  | termParams (Fun(_,ts), pairs)  = foldr termParams pairs ts
  | termParams (_, pairs)          = pairs;

val goalParams = accumGoal (accumForm termParams);
end;

```

图10-2 (续)

调用`replace(u1, u2) t`将把项`t`中所有的项`u1`替换成`u2`。这个函数由`abstract`和`subst`调用。

算子`accumForm`和`accumGoal`演示了高阶程序设计。假如对于某个类型 τ , f 具有类型 $term \times \tau \rightarrow \tau$, 其中 $f(t, x)$ 在 x 中积累了关于 t 的某些信息。(例如, f 可以是`termVars`, 它积累了项的自由变量列表。) `foldr f` 则将 f 推广到项的列表上。函数`accumForm f`具有类型 $form \times \tau \rightarrow \tau$, 将 f 推广为公式操作。这个函数用`foldr f`去处理谓词 $P(t_1, \dots, t_n)$ 的参数, 它还递归地调用`foldr (accumForm f)`去处理连接词的公式表。算子`accumGoal`两次调用`foldr`, 将一个具有类型 $form \times \tau \rightarrow \tau$ 的函数推广到一个具有类型 $(form\ list \times form\ list) \times \tau \rightarrow \tau$ 的函数。它可以将一个有关公式的函数推广到有关目标的函数。

算子`accumForm`和`accumGoal`为遍历公式和目标提供了一种统一的方式。它们定义了函数`goalVars`和`goalParams`, 并且还可以有很多类似的应用。此外, 它们是高效的: 它们不创建任何表和其他数据结构。

函数`termVars`和`termParams`是通过递归定义的, 它递归地扫描一项来积累其中的变量或参数。这两个函数都用`foldr`来遍历参数表。函数`insert` (为了节省版面略去了) 建立一个无重复字符串的有序表。注意, `termVars`不认为参数 $b_{?a_1, \dots, ?a_k}$ 包含变量 $?a_1, \dots, ?a_k$, 这些禁止变量在逻辑上不是该项的一部分, 也许应该放在一个单独的表格中。

练习 10.13 概述如何修改`FOL`和`Fol`以采用一种新的表示项的方法。约束变量使用名字标识, 但从语法上与参数和元变量是有区别的。这样的表示法能用于 λ -演算吗?

练习 10.14 改变类型`form`的声明, 将`Conn`换成为每个连接词单独设置的构造子, 比方说`Neg`、`Conj`、`Disj`、`Imp`、`Iff`。相应地修改`FOL`和`Fol`。

练习 10.15 函数`accumGoal`实际上比上面提到的更为多态。它最为一般性的类型是什么?

10.6 分析和显示公式

我们的语法分析器和美化打印程序 (分别出自第9章和第8章) 可以实现一阶逻辑的语法。

我们使用下面的文法来定义项 (*Term*)、可选参数表 (*TermPack*) 和非空项表 (*TermList*):

$$\begin{aligned} \text{TermList} &= \text{Term} \{ , \text{Term} \} * \\ \text{TermPack} &= (\text{TermList}) \\ &\quad | \text{Empty} \\ \text{Term} &= \text{Id TermPack} \\ &\quad | ? \text{Id} \end{aligned}$$

公式 (*Form*) 是通过和基本式 (*Primary*) 相互递归来定义的, 基本式由原子公式和它们的否定构成:

$$\begin{aligned} \text{Form} &= \text{ALL Id . Form} \\ &\quad | \text{EX Id . Form} \\ &\quad | \text{Form Conn Form} \\ &\quad | \text{Primary} \\ \text{Primary} &= \sim \text{Primary} \\ &\quad | (\text{Form}) \\ &\quad | \text{Id TermPack} \end{aligned}$$

409
411

量词写成ASCII字符形式时变成ALL和EX, 下面的表格给出了连接词的写法:

常用写法:	\neg	\wedge	\vee	\rightarrow	\leftrightarrow
ASCII写法:	~	&		-->	<->

由于ASCII没有希腊字母, 公式 $\exists z. \phi(z) \rightarrow \forall x. \phi(x)$ 可能要被写成

EX z. P(z) --> (ALL x. P(x))

当量化公式作为一个连接词的操作数时, Hal要求它被括在括号中。

语法分析。语法分析的签名是很短的。它只是描述了函数*read*, 用于将字符串转换为公式:

```
signature PARSE_FOL =
  sig
    val read: string -> Fol.form
  end;
```

在我们实现这个签名之前, 必须为一阶逻辑的词法分析和语法分析创建结构。结构*FolKey*定义了词法规则。之后应用第9章所述的函子:

```
structure FolKey =
  struct val alphas = ["ALL", "EX"]
        and symbols = ["(", ")", ".", " ", "?", "~",
                        "&", "|", "<->", "-->", "|-"]
  end;
structure FolLex = Lexical (FolKey);
structure FolParsing = Parsing (FolLex);
```

图10-3给出了相应的结构。它相当简单,不过有几点值得注意。

```

structure ParseFol : PARSE_FOL =
  struct
    local

      open FolParsing
      fun list ph =      ph -- repeat ("," $-- ph)    >> (op::);

      fun pack ph =      "(" $-- list ph -- "$"      >> #1
                        || empty;

      fun makeQuant ((qnt,b),p) =
        Fol.Quant(qnt, b, Fol.abstract 0 (Fol.Fun(b,[])) p);

      fun makeConn a p q = Fol.Conn(a, [p,q]);
      fun makeNeg p       = Fol.Conn("~", [p]);

      fun term toks =
        (   id -- pack term                >> Fol.Fun
          || "?" $-- id                    >> Fol.Var   ) toks;

      fun form toks =
        (   $"ALL" -- id -- "." $-- form    >> makeQuant
          || $"EX"  -- id -- "." $-- form    >> makeQuant
          || infixes (primary, Fol.precOf, makeConn) ) toks
      and primary toks =
        (   "~" $-- primary                >> makeNeg
          || "(" $-- form -- "$"          >> #1
          || id -- pack term              >> Fol.Pred   ) toks;

      in
        val read = reader form
      end
    end;
  end;

```

图10-3 一阶逻辑的语法分析

函数`list`和`pack`表示了文法短语`TermList`和`TermPack`。它们具有一般性,足以定义任意短语的“表”和“包”。

该语法分析器不能区分参数中的常量,或检测函数是否具有正确数目的实际参数:它没有保存一阶逻辑函数和谓词的任何信息。该语法分析器将任何标识符都当作常量,用`Fun("x",[])`来表示 x 。当分析量化结构 $\forall x.\phi(x)$ 时,它对量化体 $\phi(x)$ 中的“常量” x 的所有出现进行抽象。

和上一章中讨论的一样,我们的语法分析器不能接受左递归文法规则,例如

$Form = Form Conn Form$

而是依赖连接词的优先级。语法分析器调用了语法分析函数`infixes`,它有以下三个参数:

- `primary`分析连接词的操作数。
- `precOf`定义连接词的优先级。
- `makeConn`将一个连接词应用到两个公式上。

结构体的大部分都通过`local`声明设成私有。在结尾,它定义了仅有的可见标识符`read`。如果需要,为项声明一个读取函数也很容易。

显示。签名`DISPLAY_FOL`描述了公式和目标(也就是相继式)的美化打印操作符:

```
signature DISPLAY_FOL =
  sig
    val form: Fol.form -> unit
    val goal: int -> Fol.goal -> unit
  end;
```

函数`goal`的整数参数将显示在目标之前,表示子目标序号,一个证明状态通常具有几个子目标。10.14节中的会话描绘了这个输出。

结构`DisplayFol`实现了这一签名,见图10-4。必须为美化打印程序提供描写格式的符号表达式。函数`enclose`将一个表达式包在括号内,而`list`在表达式列表的元素间插入逗号。两者结合将参数表格式化成 (t_1, \dots, t_n) 。

```
structure DisplayFol : DISPLAY_FOL =
  struct
    fun enclose sexp = Pretty.blo(1, [Pretty.str(" ", sexp, Pretty.str")"]);
    fun commas [] = []
      | commas (sexp::sexps) = Pretty.str", " :: Pretty.brk 1 ::
        sexp :: commas sexps;
    fun list (sexp::sexps) = Pretty.blo(0, sexp :: commas sexps);
    fun term (Fol.Param(a,_)) = Pretty.str a
      | term (Fol.Var a) = Pretty.str ("?"^a)
      | term (Fol.Bound i) = Pretty.str "??UNMATCHED INDEX??"
      | term (Fol.Fun (a,ts)) = Pretty.blo(0, [Pretty.str a, args ts])
    and args [] = Pretty.str""
      | args ts = enclose (list (map term ts));
    fun formp k (Fol.Pred (a,ts)) = Pretty.blo(0, [Pretty.str a, args ts])
      | formp k (Fol.Conn("~", [p])) =
        Pretty.blo(0, [Pretty.str "~", formp (Fol.precOf "~") p])
      | formp k (Fol.Conn(C, [p,q])) =
        let val pf = formp (Int.max(Fol.precOf C, k))
            val sexp = Pretty.blo(0, [pf p, Pretty.str(" " ^ C),
              Pretty.brk 1, pf q])
        in if (Fol.precOf C <= k) then (enclose sexp) else sexp
        end
      | formp k (Fol.Quant(qnt,b,p)) =
        let val q = Fol.subst 0 (Fol.Fun(b,[])) p
            val sexp = Pretty.blo(2, [Pretty.str(qnt ^ " " ^ b ^ ". "),
              Pretty.brk 1, formp 0 q])
        in if k>0 then (enclose sexp) else sexp
        end
      | formp k _ = Pretty.str"??UNKNOWN FORMULA??";
    fun formList [] = Pretty.str"empty"
      | formList ps = list (map (formp 0) ps);
    fun form p = Pretty.pr (TextIO.stdOut, formp 0 p, 50);
    fun goal (n:int) (ps,qs) =
      Pretty.pr (TextIO.stdOut,
        Pretty.blo (4, [Pretty.str(" " ^ Int.toString n ^ ". "),
          formList ps, Pretty.brk 2, Pretty.str"| - ",
          formList qs]),
        50);
  end;
```

图10-4 一阶逻辑的美化打印

参数名会被打印出来,但是它的禁止变量表不会被打印。程序的另一部分将这一信息显示为一个表格。

连接词的优先级控制着括号的使用。调用`form p k q`将对 q 进行格式化,如果需要,会用括号将其括住,以便和相邻的优先级为 k 的连接词隔离开来。在生成字符串 $q \& (p \mid r)$ 时, $p \mid r$ 就被括在括号中,因为相邻的连接词($\&$)优先级为3,而 \mid 的优先级为2。

练习 10.16 解释`ParseFol`中,提供给`>>`的各个函数的作用。

练习 10.17 修改语法分析器以接受 $q \rightarrow \text{ALL } x. p$ 作为 $q \rightarrow (\forall x.p)$ 的正确语法。使得它不再要求给量化公式加上括号。

练习 10.18 在 $q \& (p1 \rightarrow (p2 \mid r))$ 中,内层的那一对括号是多余的,因为 \mid 的优先级比 \rightarrow 高,我们的美化打印经常会包含这样无用的括号。提出函数`form`的修改办法来防止这一现象。

练习 10.19 解释量化公式是怎样显示的。

10.7 合一

Hal试图将目标中的原子公式合一。它的基本合一算法接受不含有约束变量的项。给定两个项,它计算出一个(变量,项)的替换集合来把它们变为相同,或者报告这两个项不能合一。进行替换的过程称为实例化(instantiation)。合一包括三种情形:

函数应用。两个函数应用能被合一仅当它们应用的是同一个函数,很显然没有实例化可以将 $f(?a)$ 和 $g(b, ?c)$ 变换为相同的项。要将 $g(t_1, t_2)$ 和 $g(u_1, u_2)$ 合一,就要一致地合一 t_1 和 u_1 以及 t_2 和 u_2 ——也就是说, $g(?a, ?a)$ 不可能与 $g(b, c)$ 合一,因为一个变量($?a$)不可能替换成两个不同的常量(b 和 c)。

$f(t_1, \dots, t_n)$ 与 $f(u_1, \dots, u_n)$ 的合一从 t_1 与 u_1 的合一开始,然后将得到的替换应用到余下的项上。下一步自是合一 t_2 与 u_2 ,并将新的替换应用到余下的项上,依此类推。如果其中任何一次合一失败,那么该函数应用都不能合一。对应的实际参数可以以任意顺序进行合一,对结果没有实质影响。

参数。两个参数仅当具有相同的名字时才能合一。参数不能与函数应用合一。

变量。剩下的也是最有意思的情形就是将变量 $?a$ 与项 t (有别于 $?a$)合一。如果 $?a$ 没有出现在 t 中,则合一成功,产生替换 $(?a, t)$ 。如果 $?a$ 的确出现在 t 中,则合一失败——可能出于以下两个不同的原因:

- 如果 $?a$ 出现在 t 的一个参数中,那么 $?a$ 是该参数的一个“禁止变量”,也就是 t 的禁止变量。将 $?a$ 替换成 t 会违反某个量词规则的限制条件。
- 如果 t 真的包含 $?a$,那么这两项就不可能合一:不存在可以包含它自身的项。例如,不存在替换可以将 $f(?a)$ 和 $?a$ 变换成相同的项。

这就是臭名昭著的`出现检测`(occurs check),因为其开销,大多数Prolog解释器都省略它。对于定理证明来说,正确性必须优先于效率,必须进行出现检测。

例子。要将 $g(?a, f(?c))$ 与 $g(f(?b), ?a)$ 合一,首先要将 $?a$ 与 $f(?b)$ 合一,这是很明显的一步。在将余下实际参数中的 $?a$ 替换成 $f(?b)$ 之后,将 $f(?c)$ 与 $f(?b)$ 合一。这将 $?c$ 替换成 $?b$ 。结果可以写作集合 $\{?a \mapsto f(?b), ?c \mapsto ?b\}$ 。合一后的公式是 $g(f(?b), f(?b))$ 。

这里还有一个例子。要将 $g(?a, f(?a))$ 与 $g(f(?b), ?b)$ 合一，第一步还是将 $?a$ 替换成 $f(?b)$ 。下一步要做的就是将 $f(f(?b))$ 与 $?b$ 合一——这是不可能的，因为 $f(f(?b))$ 包含 $?b$ 。合一失败。

参数的实例化。我们知道每个参数都带有一个禁止变量表， $b_{?a}$ 绝对不能作为用于替换 $?a$ 的项 t 的一部分。当进行一个合法的替换时， $b_{?a}$ 中的 $?a$ 会被替换成 t 中的变量，而不是 t 本身。例如，将 $?a$ 替换成 $g(?c, f(?d))$ 使得 $b_{?a}$ 变换为 $b_{?c, ?d}$ 。任何对 $?c$ 或 $?d$ 的替换事实上也是对 $?a$ 的替换，因此，对该参数来说 $?c$ 和 $?d$ 也是被禁止的。

例如，要将 $g(?a, f(b_{?a}))$ 与 $g(h(?c, ?d), ?c)$ 合一，第一步是将 $?a$ 替换成 $h(?c, ?d)$ 。 g 的第二个实际参数就成为 $f(b_{?c, ?d})$ 和 $?c$ ，这两项是不可合一的，因为 $?c$ 对于参数 $b_{?c, ?d}$ 是被禁止的。

❶ Skolem函数。参数在定理证明中的使用并不广泛，更为传统的是Skolem函数。规则 \forall :left和 \exists :right可以引入项 $b(?a_1, \dots, ?a_k)$ 来取代参数 $b_{?a_1, \dots, ?a_k}$ 。这里， b 是一个函数符号，它不会出现在证明的其他地方。这种项和参数的作用一样，出现检测防止合一违反规则的限制条件。Skolem函数在自动证明过程中有长处，但是它破坏了公式的可读性，在高阶逻辑中它甚至会导致错误的论证。

ML代码。签名描述了合一和实例化函数，同时还描述了用于报告不可合一项的异常Failed:

```
signature UNIFY =
  sig
    exception Failed
    val atoms      : Fol.form * Fol.form -> Fol.term StringDict.t
    val instTerm    : Fol.term StringDict.t -> Fol.term -> Fol.term
    val instForm    : Fol.term StringDict.t -> Fol.form -> Fol.form
    val instGoal    : Fol.term StringDict.t -> Fol.goal -> Fol.goal
  end;
```

417

函数`atoms`试图将两个原子公式合一，而`instTerm`、`instForm`和`instGoal`则相应地将替换应用到项、公式和目标上。

我们通过一个字典来表示替换集合，字典用到了结构`StringDict` (7.10节)，变量名恰好是字符串。

原子公式由一个应用到实际参数表的谓词构成，就像 $P(t_1, \dots, t_n)$ 。两个原子公式合一与两个函数应用合一本质上是一样的，谓词必须相同，且相应的各对实际参数必须同时可合一。

结构`Unify` (图10-5)实现了合一。关键的几个函数都在`unifyLists`中进行了声明，以便可以访问`env`，它是替换的环境。在`env`中收集替换要比在每个替换生成时立即应用更有效率。替换应被看作是逐步积累起来的，而不是联立发生的，这一点如同在 λ -演算解释器中对定义的处理一样。根据

$$\{?b \mapsto g(z), ?a \mapsto f(?b)\}$$

进行联立替换会将 $?a$ 换成 $f(?b)$ ，但是我们的函数将 $?a$ 替换成 $f(g(z))$ 。这是对我们的合一算法的正确处理。

在`unifyLists`中声明了几个函数，下面是它们的一些注解：

- `chase t`当 t 是变量时，将它替换成在`env`中对它的赋值。非变量项原样返回，每一阶段，合一都只关心项的外层形式。
- `occurs a t`测试变量 $?a$ 是否在项 t 中出现，和`chase`一样，它在环境中查找变量。

- *occs1 a ts* 测试变量 *a* 是否在项表 *ts* 中出现。
- *unify(t, u)* 通过合一 *t* 与 *u*，如果可能的话，来创建一个新环境，否则抛出异常 *Failed*。如果 *t* 和 *u* 是变量，那么它们在 *env* 中一定不能事先有赋值，违反这一条件会导致一个变量拥有两个赋值！
- *unifyl(ts, us)* 同时合一表 *ts* 和 *us* 中对应的成员，如果两个表长度不等则抛出异常 *Failed*。（如果两项不可合一，则异常将由 *unify* 抛出，而不是由 *unifyl* 抛出。）

```

structure Unify : UNIFY =
  struct
    exception Failed;
    fun unifyLists env =
      let fun chase (Fol.Var a) = (chase (StringDict.lookup (env, a))
                                     handle StringDict.E _ => Fol.Var a)
          | chase t = t
      in fun occurs a (Fol.Fun(_, ts)) = occurs a ts
          | occurs a (Fol.Param(_, bs)) = occurs a (map Fol.Var bs)
          | occurs a (Fol.Var b) =
              (a=b) orelse (occurs a (StringDict.lookup (env, b))
                             handle StringDict.E _ => false)
          | occurs a _ = false
          and occsl a = List.exists (occurs a)
          and unify (Fol.Var a, t) =
              if t = Fol.Var a then env
              else if occurs a t then raise Failed
                   else StringDict.update (env, a, t)
          | unify (t, Fol.Var a) = unify (Fol.Var a, t)
          | unify (Fol.Param(a, _), Fol.Param(b, _)) = if a=b then env
                                                         else raise Failed
          | unify (Fol.Fun(a, ts), Fol.Fun(b, us)) = if a=b then unifyl(ts, us)
                                                         else raise Failed
          | unify _ = raise Failed
          and unifyl ([], []) = env
          | unifyl (t::ts, u::us) = unifyLists (unify (chase t, chase u)) (ts, us)
          | unifyl _ = raise Failed
      in unifyl end

    fun atoms (Fol.Pred(a, ts), Fol.Pred(b, us)) =
        if a=b then unifyLists StringDict.empty (ts, us) else raise Failed
        | atoms _ = raise Failed;

    fun instTerm env (Fol.Fun(a, ts)) = Fol.Fun(a, map (instTerm env) ts)
        | instTerm env (Fol.Param(a, bs)) =
            Fol.Param(a, foldr Fol.termVars [] (map (instTerm env o Fol.Var) bs))
        | instTerm env (Fol.Var a) = (instTerm env (StringDict.lookup (env, a))
                                       handle StringDict.E _ => Fol.Var a)
        | instTerm env t = t;

    fun instForm env (Fol.Pred(a, ts)) = Fol.Pred(a, map (instTerm env) ts)
        | instForm env (Fol.Conn(b, ps)) = Fol.Conn(b, map (instForm env) ps)
        | instForm env (Fol.Quant(qnt, b, p)) = Fol.Quant(qnt, b, instForm env p);

    fun instGoal env (ps, qs) = (map (instForm env) ps, map (instForm env) qs);
  end;

```

图10-5 合一

这个实现是纯函数式的。将变量表示为引用可能更为高效——更新一个变量即可完成一个替

换，不需要环境——但这和策略定理证明不相容。将一个策略应用于一个证明状态应得到新的状态，不改变原有的状态，以便可以尝试其他策略。合一算法可以使用命令式技术，条件是它们对外部不可见。

当两项不可合一时合一函数抛出异常*Failed*。就像语法分析中的那样，失败可能在深层嵌套的递归调用中被发现，这时异常将向外传播。这是一种适合异常应用的典型情况。

函数*instTerm*如前所述地在参数中进行替换。每一个禁止变量都被换成替换结果项中的一列变量。这可以用*List.concat*来完成，但是使用*foldr*和*termVars*的组合将减少复制的次数。

❶ 高效的合一算法。这里给出的算法在非常特别的情况下需要指数时间。实际情况中，它相当好用。也存在更为高效的算法。Paterson和Wegman (1978) 的线性时间算法在实践中被认为过于复杂。Martelli和Montanari (1982) 的算法几乎是线性的，是为实用而设计的。然而，Corbin和Bidoit (1983) 提出了一个基于朴素想法的算法，它用图（实际上是指针）来表示项，而不是用树。他们声称这个算法比那个几乎线性的算法还要好，因为其简单性，尽管它需要平方时间。Ružička和Prívara (1988) 将这一方案求精，也几乎达到线性的了。

练习 10.20 如果从*unify*中省去这一行将发生什么？

```
if t = Fol.Var a then env else
```

策略和证明状态

我们关于相继式演算的证明步骤是通过逐步求精来操作的，从一个目标反向进行。证明树从根向上生长。在ML中编码这一过程需要一个证明状态 (proof state) 的数据结构，这是部分构造的证明。推理规则实现为证明状态上的函数，称为策略 (tactic)。

10.8 证明状态

形式证明是一棵树，它的每一个结点都带有一个相继式和一个规则名。每个结点的分支指向其规则的前提。但是ML中与这种树相对应的数据类型并不适合我们的目的。反向证明需要访问叶子，而不是根。扩展证明将把一个叶子转换成一个分支结点，并需要复制一部分树。中间节点在证明的搜索过程中没什么用处。

Hal完全省略了中间结点。部分证明树只包含证明的两个部分。根，或主目标 (main goal)，是我们首先设定要证明的公式。叶子，或当前子目标 (current subgoal)，是那些还没有证明的相继式。

目标 ϕ 和单元素子目标表 $[\vdash \phi]$ 组成的序偶表示了 ϕ 的证明的初始状态，尚未对其应用任何规则。目标 ϕ 和空子目标表组成的序偶是结束状态，表示已完成的证明。

若不保存完整的证明树，我们如何确定Hal的证明是正确的呢？答案是用一个抽象类型*state*来隐藏证明状态的具体表示，并提供有限的操作——创建初始状态，检查状态内容，测试结束状态，并通过某个推理规则将一个状态变换到另一个状态。

如果需要更高的可靠性，那么可以将证明保存起来供另一个程序检测。要记住的是，真正的定理证明可能会非常庞大，再多的机器检测也不能提供绝对的可靠性。我们的程序和证明系统都难免会出错，就像将“现实世界”的任务简化为逻辑所使用的理论一样。

❶ 推理系统的形式化方案。在开发爱丁堡LCF的过程中，Robin Milner产生了将推理系统定义为抽象类型的构想。他设计了ML的类型系统以支持这种应用。LCF的类型 thm 代表了逻辑定理的集合。以 thm 作为返回类型的函数则实现了公理和推理规则。

将推理规则实现为从定理到定理的函数，这种方法支持正向证明，是LCF的主要论证方式。为了支持反向证明，LCF提供了策略。LCF策略通过一个类型为 $thm\ list \rightarrow thm$ 的函数表示了部分证明。当提供了关于各个子目标的定理后，这个函数可以通过推理规则证明主目标。一个完成的证明只要接受一个空表就可以证明主目标。有关的原始描述（Gordon等，1979）已经绝版了，但是我的一本关于LCF的书也描述了这方面的工作（Paulson，1987）。

Hal和LCF的不同在于它将推理规则实现为证明状态上的函数，而不是定理上的函数。这些函数本身就是策略，并支持反向证明。他们不支持正向证明。这一方案还支持合一，策略可以更新证明状态中的元变量。

Isabelle（Paulson，1994）使用了另一个方案。规则和证明状态在类型化 λ -演算中具有相同的表示法。将这些对象组合既可产生正向证明又可产生反向证明。这需要某种形式的高阶合一（Huet，1975）。

10.9 ML签名

签名 $RULE$ 描述了证明状态的抽象类型，以及它的操作（图10-6）。每个类型为 $state$ 的值都包含了一个公式（主目标）和一个相继式表（子目标）。每个状态还包含了额外的信息以供内部使用，尽管我们从签名中看不出来。

421

```
signature RULE =
  sig
    type state
    type tactic = state -> state ImpSeq.t
    val main    : state -> Fol.form
    val subgoals : state -> Fol.goal list
    val initial  : Fol.form -> state
    val final    : state -> bool
    val basic    : int -> tactic
    val unify    : int -> tactic
    val conjL    : int -> tactic
    val conjR    : int -> tactic
    val disjL    : int -> tactic
    val disjR    : int -> tactic
    val implL    : int -> tactic
    val implR    : int -> tactic
    val negL     : int -> tactic
    val negR     : int -> tactic
    val iffL     : int -> tactic
    val iffR     : int -> tactic
    val allL     : int -> tactic
    val allR     : int -> tactic
    val exL      : int -> tactic
    val exR      : int -> tactic
  end;
```

图10-6 签名 $RULE$

类型 *tactic* 是函数类型

$$state \rightarrow state \text{ ImpSeq.t}$$

的缩写, 其中 *ImpSeq* 是 8.4 节给出的惰性表结构。一个策略将一个状态映射到一个可能的后续状态序列上。原始策略产生有穷序列, 通常长度为零或一。一个复杂的策略, 比如深度优先搜索, 可以产生状态的无穷序列。

函数 *initial* 创建了初始状态, 其中包含作为主目标的已知公式, 和唯一的子目标。谓词 *final* 测试一个证明状态是否为结束状态, 也就是不包含任何子目标。

签名中的其他函数都是原始策略, 它们定义了相继式演算的推理规则。稍后将引入策略算子 (*tactical*) 来对策略进行组合。

证明状态中的子目标从 1 开始编号。每个原始策略, 在给定一个整数参数 *i* 和一个状态后, 都对子目标 *i* 应用相继式演算的某个规则, 创建出一个新的状态。例如, 调用

$$\text{conjL } 3 \text{ st}$$

对状态 *st* 中的子目标 3 应用 $\wedge : \text{left}$ 。如果这一子目标形如 $\phi \wedge \psi, \Gamma \vdash \Delta$, 则后续状态的子目标 3 将是 $\phi, \psi, \Gamma \vdash \Delta$ 。否则, $\wedge : \text{left}$ 不能应用到该子目标上, 也就没有后续状态, 这时 *conjL* 会返回空序列。

如果 *st* 的子目标 5 是 $\Gamma \vdash \Delta, \phi \wedge \psi$, 那么

$$\text{conjR } 5 \text{ st}$$

将构造一个新的状态, 其子目标 5 是 $\Gamma \vdash \Delta, \phi$, 子目标 6 是 $\Gamma \vdash \Delta, \psi$ 。在原来的 *st* 中编号大于 5 的子目标编号将向后移。

调用 *basic i st* 检测状态 *st* 中的子目标 *i* 是否为基本相继式。如果子目标 *i* 中两边有相同的公式, 则将该子目标从后续状态中删去。否则, 这一策略返回空序列以示失败。对基本相继式更为精细的处理是策略 *unify*。

调用 *unify i st* 试图通过将状态 *st* 中的子目标 *i* 转换为一个基本相继式来对子目标 *i* 进行求解。如果该策略能将左侧的一个公式与右侧的一个公式合一, 那么它将删除子目标 *i*, 并将合一得出的替换应用到证明状态的剩余部分。有可能存在几个不同的可合一的公式对, 将 *unify* 应用到子目标

$$P(?a), P(?b) \vdash P(f(c)), P(c)$$

产生四个后续状态的序列。只有第一个会被计算, 其他的只在需要时才进行计算, 这是因为序列是惰性的。

10.10 用于基本相继式的策略

结构 *Rule* 是分几部分给出的。第一部分 (图 10-7) 定义了类型 *state* 的表示法, 及其基本操作, 并且声明了用于基本相继式的策略。

声明类型 *state*。datatype 声明引入了类型 *state* 和它的构造子 *State*。该构造子不被输出, 使得只能在结构体内访问类型的具体表示。声明类型 *tactic* 来缩写从状态到状态序列的函数类型。

函数 *main* 和 *subgoals* 返回了证明状态的相应部分。证明状态的第三个分量是一个整数, 用于生成量词规则中的具唯一性的名字。它的值从 0 开始, 并在后续状态创建时按需要递增。如

果这一名字计数器存放在一个引用单元中，并用赋值来更新，那么许多代码会得到简化——特别是在没有用到计数器的地方。然而，将量词规则应用到一个状态上时，会影响共享那个引用的所有状态。将计数器清0，虽然可以生成较短的名字，也可能导致名字被重用以及错误的论证。最安全的还是保证所有策略都是纯函数式的。

```

structure Rule :> RULE =
  struct
    datatype state = State of Fol.goal list * Fol.form * int

    type tactic = state -> state ImpSeq.t;

    fun main      (State(gs,p,_)) = p
    and subgoals  (State(gs,p,_)) = gs;

    fun initial p = State([ ([], [p]) ], p, 0);

    fun final (State(gs,_,_)) = null gs;

    fun spliceGoals gs newgs i = List.take(gs,i-1) @ newgs @ List.drop(gs,i);

    fun propRule goalF i (State(gs,p,n)) =
      let val gs2 = spliceGoals gs (goalF (List.nth(gs,i-1))) i
      in ImpSeq.fromList [State(gs2, p, n)] end
      handle _ => ImpSeq.empty;

    val basic = propRule
      (fn (ps,qs) =>
        if List.exists (fn p => List.exists (fn q => p=q) qs) ps
        then [] else raise Match);

    fun unifiable ([], _) = ImpSeq.empty
      | unifiable (p::ps, qs) =
        let fun find [] = unifiable (ps,qs)
          | find (q::qs) = ImpSeq.cons(Unify.atoms(p,q), fn()=> find qs)
          handle Unify.Failed => find qs
        in find qs end;

    fun inst env (gs,p,n) =
      State (map (Unify.instGoal env) gs, Unify.instForm env p, n);

    fun unify i (State(gs,p,n)) =
      let val (ps,qs) = List.nth(gs,i-1)
        fun next env = inst env (spliceGoals gs [] i, p, n)
      in ImpSeq.map next (unifiable(ps,qs)) end
      handle Subscript => ImpSeq.empty;
  end

```

图10-7 Rule的第一部分——用于基本相继式的策略

调用`initial p`创建一个状态仅包含相继式 $\vdash p$ 作为它唯一的子目标，并且将 p 作为主目标，0作为变量计数器。谓词`final`用来测试子目标表是否为空。

`basic`和`unify`的定义。所有策略都是用函数`spliceGoals`来表示的，它将一个状态中的子目标 i 替换成一系列新的子目标。`List`的函数`take`和`drop`分别用于提取出 i 之前和 i 之后的子目标，使得那些新的子目标可以被拼接在正确的位置上。


`propRule`的声明说明了证明状态是怎样被处理的。这个函数根据类型为 $goal \rightarrow goal\ list$ 的

函数`goalF`生成一个策略。将该策略应用到整数 i 和状态上时，它将子目标 i 提供给`goalF`并将其结果拼接回去；它将新状态作为一个单元素序列返回。如果其中出现异常，它将返回一个空序列。如果不存在第 i 个子目标，调用`List.nth(gs, i-1)`会抛出异常`Subscript`，要注意函数`nth`对表元素是从零开始编号的。其他异常，如`Match`，可能会从`goalF`中抛出。

策略`basic`是`propRule`的简单应用。它将一个测试目标 (ps, qs) 是否为基本相继式的函数作为`goalF`传入。如果是，则返回一个空的子目标表，其效果就相当于从后续状态中删除原先的子目标。但如果 (ps, qs) 不是一个基本相继式，则函数抛出异常。

策略`unify`则更为复杂：它能返回多个后续状态。它调用`unifiable`来生成合一环境的序列，并调用`inst`将它们应用到其他子目标上。函数`next`进行最后的处理，它是通过算子`ImpSeq.map`来应用的。

函数`unifiable`接受公式表 ps 和 qs 。将 ps 中的某些 p 与 qs 中的某些 q 进行合一，由函数`unifiable`返回得到的所有环境序列。函数`find`处理“内循环”，在 qs 中搜索元素与 p 合一。它产生一个序列，其首元素是一个环境，其尾部将由递归调用`find qs`生成，但是若`Unify.atoms`抛出异常，则结果只是`find qs`。

 注意其他目标。当`unify`解决了一个子目标后，它可能会更新状态，使得其他子目标变得不可证明。这一策略的成功并不保证它是找到一个证明的正确途径，在某些情形中，要换用另一种不同的策略。任何涉及`unify`的搜索过程都应使用回溯。另一方面，通过`basic`解决一个目标总是安全的。

练习 10.21 给出一个例子来证明上面的警告是有道理的。

10.11 命题策略

`Rule`的下一部分实现了 \wedge 、 \vee 、 \neg 、 \rightarrow 和 \leftrightarrow 的规则。由于每个连接词都有“左”规则和“右”规则，所以总共有10个策略。见图10-8。

这些策略使用了同样的基本机制。在给定的左侧或右侧搜索适合的公式，取下其中的连接词，根据连接词的操作数产生新的子目标。每个策略如果成功都返回单一的后续状态。如果找不到适合的公式，则策略失败，返回空的状态序列。在`propRule`和另一个新函数`splitConn`的辅助下，我们可以简明地表达这些策略。

举例来说明`splitConn`是怎样工作的。给定字符串`"&"`和公式表 qs ，这个函数将寻找匹配`Conn("&", ps)`的第一个元素，如果找不到就抛出异常`Match`。它还复制除去匹配元素外的 qs 。它返回 ps 和缩短的 qs 。注意，现在 ps 包含了被选出的公式的操作数。

算子`propL`辅助表示相继式规则。对给定的相继式，它将在其左侧搜索连接词。并将`splitConn`的调用结果提供给另一个负责创建新子目标的函数`leftF`。算子`propR`与之类似，只不过是在右侧搜索。

这些策略由`val`声明给出，因为它们没有显式的参数。每个策略由一个`propL`或`propR`的调用构成。每个调用在`fn`记法中传入参数`leftF`或`rightF`。每个函数接受一个分解出来的子目标并返回一个或两个子目标。由此，`conjL`在左侧搜索一个合取式，并将两个合取项插入新的子目标，而`conjR`则在右侧搜索一个合取式，并生成两个子目标。

```

fun splitConn a qs =
  let fun get [] = raise Match
      | get (Fol.Conn(b,ps) :: qs) = if a=b then ps else get qs
      | get (q::qs) = get qs;
      fun del [] = []
      | del ((q as Fol.Conn(b,_)) :: qs) = if a=b then qs
                                          else q :: del qs
      | del (q::qs) = q :: del qs
  in (get qs, del qs) end;

fun propL a leftF = propRule (fn (ps,qs) => leftF (splitConn a ps, qs));
fun propR a rightF = propRule (fn (ps,qs) => rightF (ps, splitConn a qs));
val conjL = propL "&" (fn ([p1,p2], ps), qs) => [(p1::p2::ps, qs)];
val conjR = propR "&"
  (fn (ps, ([q1,q2], qs)) => [(ps, q1::qs), (ps, q2::qs)]);
val disjL = propL "|"
  (fn ([p1,p2], ps), qs) => [(p1::ps, qs), (p2::ps, qs)];
val disjR = propR "|" (fn (ps, ([q1,q2], qs)) => [(ps, q1::q2::qs)]);
val implL = propL "-->"
  (fn ([p1,p2], ps), qs) => [(p2::ps, qs), (ps, p1::qs)];
val implR = propR "-->" (fn (ps, ([q1,q2], qs)) => [(q1::ps, q2::qs)]);
val negL = propL "~" (fn ([p], ps), qs) => [(ps, p::qs)];
val negR = propR "~" (fn (ps, ([q], qs)) => [(q::ps, qs)]);
val iffL = propL "<->"
  (fn ([p1,p2], ps), qs) => [(p1::p2::ps, qs), (ps, p1::p2::qs)];
val iffR = propR "<->"
  (fn (ps, ([q1,q2], qs)) => [(q1::ps, q2::qs), (q2::ps, q1::qs)]);

```

图10-8 Rule的部分——命题策略

10.12 量词策略

上面所给出的机制可以容易地修改以表示量词策略，不过和命题的情况有几点不同。代码在图10-9中，加上它就完成了整个Rule结构。

函数splitQuant与splitConn非常类似。它用来寻找第一个含有指定量词("ALL"或"EX")的公式。它返回整个公式(而不是只有操作数)，因为某些量词策略将在子目标中保留它。

虽然我们的相继式演算是用多重集合定义的，但它却是用表实现的。相继式中的公式是有顺序的，如果表中有两个满足要求的公式，那么将找到最左边的。为了遵守多重集合的概念，Hal不提供重新排序公式的方法。量词策略保证没有公式会永远被排除在外。

这些策略需要一个全新名字的来源，以命名变量和参数。调用letter n，其中 $0 \leq n < 25$ ，返回从"a"到"z"的单字符字符串。函数gensym——函数名出自古老的Lisp——根据自然数产生一个字符串。它的结果是一个26进制数，其中的“数字”是小写字母，前缀“_”防止和外部提供的名字冲突。

算子quantRule根据函数goalF产生一个策略。它提供一个子目标和一个新名字给goalF，goalF相应地具有类型 $goal \times string \rightarrow goal\ list$ 。当构造后续状态时，它还要递增变量计数器。

其他方面, *quantRule*和*propRule*完全一样。

```

fun splitQuant qnt qs =
  let fun get [] = raise Match
      | get ((q as Fol.Quant(qnt2,_,p)) :: qs) = if qnt=qnt2 then q
                                                else get qs
      | get (q::qs) = get qs;
      fun del [] = []
      | del ((q as Fol.Quant(qnt2,_,p)) :: qs) = if qnt=qnt2 then qs
                                                else q :: del qs
      | del (q::qs) = q :: del qs
  in (get qs, del qs) end;

fun letter n = String.substring("abcdefghijklmnopqrstuvwxyz", n, 1)

fun gensym n =
  if n<26 then "_" ^ letter n
  else gensym(n div 26) ^ letter(n mod 26);

fun quantRule goalF i (State(gs,p,n)) =
  let val gs2 = spliceGoals gs (goalF (List.nth(gs,i-1), gensym n)) i
  in ImpSeq.fromList [State(gs2, p, n+1)] end
  handle _ => ImpSeq.empty;

val allL = quantRule (fn ((ps,qs), b) =>
  let val (qntForm as Fol.Quant(_,_,p), ps') = splitQuant "ALL" ps
      val px = Fol.subst 0 (Fol.Var b) p
  in [(px :: ps' @ [qntForm], qs)] end);

val allR = quantRule (fn ((ps,qs), b) =>
  let val (Fol.Quant(_,_,q), qs') = splitQuant "ALL" qs
      val vars = Fol.goalVars ((ps,qs), [])
      val qx = Fol.subst 0 (Fol.Param(b, vars)) q
  in [(ps, qx::qs')] end);

val exL = quantRule (fn ((ps,qs), b) =>
  let val (Fol.Quant(_,_,p), ps') = splitQuant "EX" ps
      val vars = Fol.goalVars ((ps,qs), [])
      val px = Fol.subst 0 (Fol.Param(b, vars)) p
  in [(px::ps', qs)] end);

val exR = quantRule (fn ((ps,qs), b) =>
  let val (qntForm as Fol.Quant(_,_,q), qs') = splitQuant "EX" qs
      val qx = Fol.subst 0 (Fol.Var b) q
  in [(ps, qx :: qs' @ [qntForm])] end);

end;

```

图10-9 Rule的最后部分——量词策略

每个策略都表示为*quantRule*对一个函数的应用, 该函数使用fn记法, 它接受子目标(*ps*, *qs*)和一个新名字*b*, 并返回一个子目标。

策略*allL*和*exR*用于展开一个量化公式。它们将一个名字为*b*的变量替换进公式体。它们包括子目标中的量化公式(通过*as*绑定到*qntForm*)。公式被放在表的最后, 以便其他量化公式可以在下一次策略应用时被选上。

策略*allR*和*exL*选择一个量化公式并将一个参数替换进公式体。该参数名字为*b*, 并带有子目标中的所有变量作为禁止变量。

当我们读到Rule结尾时应该记得，其中声明的策略只是创建类型state值的方法。所有证明过程——即便它们通过深奥的数据结构说明了其有效性——最终必须应用这些策略来构造一个形式证明。如果上面给出的代码是正确的，并且ML系统是正确的，那么Hal的证明就可以保证是正确的。从此之后的编码错误都不会导致错误的证明。这种可靠性是由于我们已将state定义为一个抽象类型。

练习 10.22 建议一种可以存储整个证明树的类型state的表示法。最好是使用一种编码，在不占多少空间的同时允许重构证明树。概述对于RULE和Rule所作出的修改。

练习 10.23 我们的这套策略没有提供在证明中使用已经证明定理的方法。基于规则

$$\frac{\vdash \phi \quad \phi, \Gamma \vdash \Delta}{\Gamma \vdash \Delta}$$

的策略可以将定理 $\vdash \phi$ 作为一个引理插入目标。^①叙述怎样实现这样一个策略。

练习 10.24 “结构Rule没有涉及ParseFol或DisplayFol，因此语法分析和美化打印中的错误不会导致错误证明的构造。”评价这一观点。

搜索证明

大部分程序设计都已经被抛在我们身后了。我们就要准备在机器上尝试证明了。我们要实现一套命令来将策略应用到一个目标上。这里将要说明如何处理证明状态，也会暴露冗长乏味的逐条规则的证明检测。策略算子，通过为策略提供控制结构，将允许我们用短短几行代码来表示自动定理证明机。

10.13 变换证明状态的命令

用户界面并不采用从终端读入数据，而是由一套命令组成的，这套命令可以在ML顶层调用。这是策略定理证明机惯用的做法。最重要的命令就是“应用一个策略”，并且该策略可以由任意一个ML表达式给出，因此，命令语言是ML本身。请记住，ML代表元语言（Meta Language）。

Hal的界面是粗糙的。它仅仅提供命令来设定、更新和检查存储证明状态。实用的定理证明需要额外的功能，例如用一个undo命令来回复到一个先前的状态。由于一个策略可能返回多个后续状态，策略的应用定义了一棵以初始状态为根的搜索树。图形用户界面会提供办法来考察这棵树。为了保持代码简单，这些功能留作练习。有些ML系统可以和像Tcl/Tk这样的脚本语言通信，使得在屏幕上放置窗口和菜单变容易了。良好的界面设计还需要对用户工作习惯的仔细研究。

签名COMMAND描述了用户界面：

```
signature COMMAND =
  sig
    val goal      : string -> unit
    val by        : Rule.tactic -> unit
    val pr        : Rule.state -> unit
```

① 这一规则是“cut”的一种特殊情形，它的第一个前提可以是 $\Gamma \vdash \Delta, \phi$ 。

```
val getState : unit -> Rule.state
end;
```

界面由下面几项组成，它们（除了 pr ）都作用在一个存储证明状态上：

- $goal$ 命令接受一个通过字符串给出的公式 ϕ ，它设置存储证明状态为 ϕ 的初始状态。
- by 命令应用一个策略到当前状态上。如果返回的后续状态序列不为空，那么它的首元素会被取出来更新存储证明状态。否则，该策略失败，显示一条错误信息。
- pr 命令打印它的参数，也就是一个证明状态，到终端上。
- 函数 $getState$ 返回存储证明状态。

结构 $Command$ 实现了这些项（图10-10）。当前状态存储在一个引用单元中，初始化为虚构的目标“No goal yet!”。

```
structure Command : COMMAND =
  struct

    val currState = ref (Rule.initial (Fol.Pred("No goal yet!", [])));

    fun question (s,z) = " ?" :: s :: z;

    fun printParam (a, []) = ()          (* 打印参数表格的一行 *)
      | printParam (a, ts) =
        print (String.concat (a :: " not in " ::
                               foldr question ["\n"] ts));

    fun printGoals (_, []) = ()
      | printGoals (n, g::gs) = (DisplayFol.goal n g; printGoals (n+1, gs));

    fun pr st = (* 打印一个证明状态 *)
      let val p = Rule.main st
          and gs = Rule.subgoals st
      in DisplayFol.form p;
        if Rule.final st then print "No subgoals left!\n"
        else (printGoals (1, gs);
              app printParam (foldr Fol.goalParams [] gs))
      end;

    (* 打印新状态，然后设它为当前状态 *)
    fun setState state = (pr state; currState := state);

    val goal = setState o Rule.initial o ParseFol.read;

    fun by tac = setState (ImpSeq.hd (tac (!currState)))
      handle ImpSeq.Empty => print "*** Tactic FAILED! **\n"

    fun getState() = !currState;
  end;
```

图10-10 用户界面命令

回忆可知，一个参数，如 b_{η_c, η_d} 只被显示为 b 。界面将每一个参数和它的禁止变量显示在一个表格中。函数 $printParam$ 打印这样一行

```
b not in ?c ?d
```

代表 b_{η_c, η_d} ，对于一个没有禁止变量的参数来说什么也不打印。函数 $printGoals$ 打印一个带编号

的子目标表。在这些函数的辅助下, *pr*将打印一个状态: 它的主目标、子目标表, 以及参数表格。

练习 10.25 设计和实现一个*undo*命令, 取消最近一次*by*命令的效果。重复*undo*命令应该回复到越来越早的状态。

练习 10.26 有很多方法可以管理状态搜索树。界面可以考察树中的一条路径。每个结点会存储一个可能的后续状态序列, 将其中一个作为活动分支。改变任一结点的活动分支都将选择一条不同的路径。完善这一思想。

10.14 两个使用策略的证明实例

为了演示这些策略和用户界面, 让我们在机器上进行一些证明。为了便于使用命令, 我们打开相应的模块:

```
open Command;
```

现在我们来证明。第一个例子很简单, 证明 $\phi \wedge \psi \rightarrow \psi \wedge \phi$ 。goal命令将这个公式提供给Hal。

```
goal "P & Q --> Q & P";
> P & Q --> Q & P
> 1. empty |- P & Q --> Q & P
```

现在 $\phi \wedge \psi \rightarrow \psi \wedge \phi$ 是主目标, 也是唯一的子目标。我们必须对子目标1应用 \rightarrow :right, 不可能有其他步骤:

```
by (Rule.impR 1);
> P & Q --> Q & P
> 1. P & Q |- Q & P
```

子目标1变为 $\phi \wedge \psi \vdash \psi \wedge \phi$, 这我们已经在纸上证明过了。虽然对这个目标可以应用 \wedge :right, 但是 \wedge :left可以得到更短的证明, 因为它只产生一个子目标。

```
by (Rule.conjL 1);
> P & Q --> Q & P
> 1. P, Q |- Q & P
```

我们又没有其他选择了, 只能对子目标1应用 \wedge :right。下面是尝试其他策略的结果:

```
by (Rule.disjR 1);
> ** Tactic FAILED! **
```

这回使用 \wedge :right了。它产生两个子目标。

```
by (Rule.conjR 1);
> P & Q --> Q & P
> 1. P, Q |- Q
> 2. P, Q |- P
```

通常都对子目标1应用策略, 让我们试试子目标2, 以增加一些变化。这是一个基本相继式, 所以它属于*Rule.basic*。

```
by (Rule.basic 2);
> P & Q --> Q & P
> 1. P, Q |- Q
```

431
433

子目标1也是个基本相继式，把它解决将终止这个证明。

```
by (Rule.basic 1);
> P & Q --> Q & P
> No subgoals left!
```

大多数定理证明机都提供一些方法来存储证明过的定理，但是在Hal中不行。我们继续看下一个例子， $\exists z.\phi(z) \rightarrow \forall x.\phi(x)$ ，之前曾讨论过。

```
goal "EX z. P(z) --> (ALL x. P(x))";
> EX z. P(z) --> (ALL x. P(x))
> 1. empty |- EX z. P(z) --> (ALL x. P(x))
```

唯一可能的步骤是对子目标1应用 \exists :right。这个策略生成一个名为? $_a$ 的变量。

```
by (Rule.exR 1);
> EX z. P(z) --> (ALL x. P(x))
> 1. empty
> |- P(?_a) --> (ALL x. P(x)),
> EX z. P(z) --> (ALL x. P(x))
```

我们可以再次应用 \exists :right，但是看上去分析子目标1中的另一个公式更为合理。因此我们应用 \rightarrow :right。

```
by (Rule.impR 1);
> EX z. P(z) --> (ALL x. P(x))
> 1. P(?_a)
> |- ALL x. P(x),
> EX z. P(z) --> (ALL x. P(x))
```

434

继续处理第一个公式，我们应用 \forall :right。这个策略生成一个名为 $_b$ 的参数，? $_a$ 为它的禁止变量。这次显示参数表格了。

```
by (Rule.allR 1);
> EX z. P(z) --> (ALL x. P(x))
> 1. EX z. P(z) --> (ALL x. P(x))
> P(?_a) |- P(_b),
> _b not in ?_a
```

由于子目标的左侧包含 $P(?_a)$ ，右侧包含 $P(_b)$ ，因此我们可以尝试通过调用 $Rule.unify$ 来合一这些公式。然而， $_b$ 的禁止变量不允许这种合一。用 $_b$ 替换? $_a$ 违反了 \forall :right的限制条件。

```
by (Rule.unify 1);
> ** Tactic FAILED! **
```

4

这个情况与开始的证明相似，只不过子目标多包含了两个新的原子公式。由于它们不能合一，我们除了再次应用 \exists :right展开量词之外没有其他选择。这次创建了变量? $_c$ 。

```
by (Rule.exR 1);
> EX z. P(z) --> (ALL x. P(x))
> 1. P(?_a)
> |- P(?_c) --> (ALL x. P(x)), P(_b),
> EX z. P(z) --> (ALL x. P(x))
> _b not in ?_a
```

证明像前面那样继续进行，并带着那两个原子公式。为了避免第三次应用 \exists :right，我们应用 \rightarrow :right。

```
by (Rule.impR 1);
> EX z. P(z) --> (ALL x. P(x))
> 1. P(?_c), P(?_a)
>   |- ALL x. P(x), P(_b),
>   EX z. P(z) --> (ALL x. P(x))
> _b not in ?_a
```

这个子目标在左侧新添了一个公式 $P(?_c)$ ，并且 $?_c$ 不是 $_b$ 的禁止变量。因此， $P(?_c)$ 与 $P(_b)$ 是可合一的。

```
by (Rule.unify 1);
> EX z. P(z) --> (ALL x. P(x))
> No subgoals left!
```

虽然第一次尝试用`Rule.unify`合一失败了，但是最后还是找到了一个成功的证明。这也说明了实际中参数和变量是如何运转的。

435

10.15 策略算子

上一节的证明实例异常地短。即使是一个简单公式的证明也需要很多步。为了说服自己，可以尝试证明

$$((\phi \leftrightarrow \psi) \leftrightarrow \chi) \leftrightarrow (\phi \leftrightarrow (\psi \leftrightarrow \chi))$$

虽然证明很长，但每一步都很明显。通常，只有一至两条规则可以应用到一个子目标上。此外，可以按任何顺序对这些子目标进行处理，因为一个成功的证明必须证明它们全部。我们总可以从子目标1着手。一个好的证明过程可以用策略表达，辅以一些控制结构。

基本策略算子。策略上的操作称为策略算子 (tactical)，类似于函数和算子。最简单的策略算子实现了顺序、选择和重复的控制结构。它们类似于语法分析操作符`--`、`||`和`repeat` (9.2节)。因此它们共享同样的名字，另外还有一个中缀操作符`|@|`。

Hal中的策略算子包含序列上的操作。类型`multifun`是签名 (图10-11) 中类型的缩写。策略算子并不限于策略。它们都是多态的，类型`state`没在任何地方出现。让我们按照这些策略算子在任意具有合适类型的函数上的作用来对它们进行讲述，而不只是针对策略。

策略算子`--`顺序地复合两个函数。当函数 $f--g$ 被应用到 x 上时，它计算序列 $f(x) = [y_1, y_2, \dots]$ ，并返回序列 $g(y_1), g(y_2), \dots$ 的连接。给定两个策略，`--`应用一个策略接着应用另一个策略到一个证明状态上，返回所有得出的“后续的后续”状态。

策略算子`||`在两个函数间选择。当函数 $f||g$ 应用到 x 上时，如果序列 $f(x)$ 非空，则返回 $f(x)$ ，否则返回 $g(x)$ 。给定两个策略，`||`将一个策略应用到一个证明状态上，如果失败，则尝试另一个策略。策略算子`|@|`提供不那么严格的一种选择，当将 $f|@|g$ 应用于 x 时，它连接序列 $f(x)$ 和 $g(x)$ 。

策略`all`和`no`可以和策略算子一同使用以得到如重复之类的效果。对于所有 x ，`all(x)`返回单元序列 $[x]$ ，而`no(x)`返回空序列。因此`all`对所有实际参数都成功，而`no`则不会成功。注意，`all`是`--`的单位元：

$$all--f = f--all = f$$

类似地, *no* 是 $||$ 和 $|@|$ 的单位元。

```

infix 5 --;
infix 0 || |@|;
signature TACTICAL =
sig
  type ('a,'b) multifun = 'a -> 'b ImpSeq.t
  val --                : ('a,'b) multifun * ('b,'c) multifun -> ('a,'c) multifun
  val ||                : ('a,'b) multifun * ('a,'b) multifun -> ('a,'b) multifun
  val |@|               : ('a,'b) multifun * ('a,'b) multifun -> ('a,'b) multifun
  val all               : ('a,'a) multifun
  val no                : ('a,'b) multifun
  val try               : ('a,'a) multifun -> ('a,'a) multifun
  val repeat            : ('a,'a) multifun -> ('a,'a) multifun
  val repeatDeterm      : ('a,'a) multifun -> ('a,'a) multifun
  val depthFirst        : ('a->bool) -> ('a,'a) multifun -> ('a,'a) multifun
  val depthlter         : ('a->bool) * int -> ('a,'a) multifun -> ('a,'a) multifun
  val firstF            : ('a -> ('b,'c) multifun) list -> 'a -> ('b,'c) multifun
end;

```

图10-11 签名TACTICAL

实现策略算子。让我们来看结构Tactical (图10-12)。序列连接所扮演的角色很清楚, 但是它在 $|@|$ 中的作用是隐晦的。这个明显的定义有什么问题呢?

```
fun (tac1 |@| tac2) x = ImpSeq.append(tac1 x, tac2 x);
```

这一版本的 $|@|$ 可能过早地 (或没必要的) 调用了 *tac2*。用 *ImpSeq.concat* 定义 $|@|$ 保证了 *tac2* 在 *tac1* 产生的元素被用完之前不会被调用。在一种惰性语言中, $|@|$ 的这个明显的定义将会正确工作。

```

structure Tactical : TACTICAL =
struct
  type ('a,'b) multifun = 'a -> 'b ImpSeq.t

  fun (tac1 -- tac2) x = ImpSeq.concat (ImpSeq.map tac2 (tac1 x));

  fun (tac1 || tac2) x =
    let val y = tac1 x
    in if ImpSeq.null y then tac2 x else y end;

  fun (tac1 |@| tac2) x =
    ImpSeq.concat (ImpSeq.cons (tac1 x, (* 延时对tac2的应用! *)
                                fn()=> ImpSeq.cons (tac2 x,
                                                         fn()=> ImpSeq.empty)));

  fun all x = ImpSeq.fromList [x];

  fun no x = ImpSeq.empty;

  fun try tac = tac || all;

  fun repeat tac x = (tac -- repeat tac || all) x;

```

图10-12 策略算子

```

fun repeatDeterm tac x =
  let fun drep x = drep (ImpSeq.hd (tac x))
        handle ImpSeq.Empty => x
      in ImpSeq.fromList [drep x] end;

fun depthFirst pred tac x =
  (if pred x then all else tac -- depthFirst pred tac) x;

fun depthIter (pred,d) tac x =
  let val next = ImpSeq.toList o tac
      fun dfs i (y, sf) () =
        if i<0 then sf()
        else if i<d andalso pred y
          then ImpSeq.cons(y, foldr (dfs (i-1)) sf (next y))
          else foldr (dfs (i-1)) sf (next y) ()
        fun deepen k = dfs k (x, fn()=> deepen (k+d)) ()
      in deepen 0 end;

fun orelseF (tac1, tac2) u = tac1 u || tac2 u;

fun firstF ts = foldr orelseF (fn _ => no) ts;
end;

```

图10-12 (续)

策略算子`try`试图应用它的参数。

策略算子`repeat`重复地应用一个函数。`repeat f x`的结果是一个序列，其中的元素值是从`x`开始通过重复应用`f`而得到的，这一重复直到进一步应用`f`导致失败为止。这个策略算子是递归定义的，就像相应的语法分析操作符。

策略算子`repeatDeterm`也提供重复。它是确定的：它只考虑每一步返回的第一个结果。当不需要其他结果时，`repeatDeterm`比`repeat`要高效得多。

策略算子`depthFirst`探索由函数生成的搜索树。调用`depthFirst pred f x`返回一个值的序列，其中的元素值都满足谓词`pred`，它们是从`x`开始通过重复应用`f`而得到的。

策略算子`depthIter`使用深度优先迭代深化来搜索树。它首先搜索到深度`d`，其次到`2d`，然后到`3d`，依此类推，这保证不会错过任何的解。函数的其他参数和`depthFirst`中的一样。它是基于5.20节所讨论的代码来实现的，相当混乱。

最后，`firstF`是一个组合原始推理规则的方便途径，见图10-13。

一些例子。为了演示这些策略算子，首先打开它们的结构，使得其中的中缀操作符可用。

```
open Tactical;
```

现在让我们来证明下面的公式，这是合取的结合律：

```

goal "(P & Q) & R --> P & (Q & R)";
> (P & Q) & R --> P & (Q & R)
> 1. empty |- (P & Q) & R --> P & (Q & R)

```

唯一可应用的规则是`→:right`。再多看一点儿，我们可以预测要应用两次`∧:left`。通过`repeat`可按需应用这两条规则：

```

by (repeat (Rule.impR 1 || Rule.conjL 1));
> (P & Q) & R --> P & (Q & R)
> 1. P, Q, R |- P & (Q & R)

```

现在必须应用两次`∧:right`。我们不断地将`Rule.basic`和相应的规则一起应用，`Rule.basic`用来

探测基本相继式:

```
by (repeat (Rule.basic 1 || Rule.conjR 1));
> (P & Q) & R --> P & (Q & R)
> No subgoals left!
```

只用了两次by命令就证明了这条定理, 如果一条条地使用规则将会需要8次命令。为了做另一个演示, 让我们来构造一个精致的策略来证明定理。还用我们熟悉的量词的例子:

```
goal "EX z. P(z) --> (ALL x. P(x))";
> EX z. P(z) --> (ALL x. P(x))
> 1. empty |- EX z. P(z) --> (ALL x. P(x))
```

我们把10.14节中使用过的那些策略放在一起进行重复, 要小心选择它们的顺序。显然应该首先尝试Rule.unify, 因为它可能一下子就将目标解决。Rule.exR必须放在最后, 否则它每次都会被应用而导致死循环。

439

```
by (repeat (Rule.unify 1 || Rule.impR 1 ||
            Rule.allR 1 || Rule.exR 1));
> EX z. P(z) --> (ALL x. P(x))
> No subgoals left!
```

❶ 策略算子简史。策略算子起源于爱丁堡LCF (Gordon等, 1979)。类似的控制结构出现在重写中 (Paulson, 1983), 用于表达称为转换 (conversion) 的重写方法。HOL系统依赖于这一方案进行重写 (Gordon和Melham, 1993, 第23章)。

在LCF和HOL中的策略算子类似于我们的语法分析操作符: 它们使用异常, 而不是返回一个结果序列。Isabelle的策略算子返回序列以便允许回溯和其他的搜索策略 (Paulson, 1994)。Hal的策略算子基本是基于Isabelle的。

传统上策略算子使用像THEN、ORELSE、REPEAT等这样的名字, 不过这违背了我们关于只有构造子以大写字母开始的约定。

练习 10.27 策略repeat(f--f) (x)有什么作用?

练习 10.28 depthFirst真的能进行深度优先搜索吗? 详细解释这个函数是怎样工作的。

练习 10.29 讲述在怎样的情况下--或|@|返回的序列会缺少一些直觉上应该存在的元素。实现没有这种缺陷的新策略算子。--和|@|有什么相应的优点作为补偿?

练习 10.30 策略算子repeat和depthFirst都是以它们的传统形式出现的。它们的效率对于交互式证明来说足够了, 但是用在证明程序中就不行了。编写更为高效的版本, 不要使用--。

10.16 一阶逻辑的自动策略

使用策略算子, 我们将编写两个简单的策略用于自动证明。给定一个子目标, depth试图通过合一、分解某个公式, 或展开量词来求解。量词可以无限地不断展开, 这个策略可能永远也停不下来。

depth中的成员本身都可用于交互式证明, 特别是当depth失败时。它们在签名TAC中描述为:

```
signature TAC =
  sig
```

440


```

val safeSteps: int -> Rule.tactic
val quant    : int -> Rule.tactic
val step     : int -> Rule.tactic
val depth    : Rule.tactic
val depthIt  : int -> Rule.tactic
end;

```

这个签名描述了五个策略:

- *safeSteps* *i* 对子目标 *i* 应用了一系列非空的“安全”规则。这些规则是除了 \exists :right 和 \forall :left 之外的其他规则。也不包括策略 *unify*, 因为它会影响其他目标。
- *quant* *i* 展开了子目标 *i* 中的量词。它有可能既应用 \exists :right 也应用 \forall :left。
- *depth* 通过深度优先搜索对所有子目标进行求解。它使用了 *safeSteps*、*unify* 和 *quant*。
- *step* *i* 尽可能使用安全步骤对子目标 *i* 进行求精, 不行的话则尝试合一以及量词展开。
- *depthIt* *d* 通过增量为 *d* 的深度优先迭代深化来对所有子目标进行求解。它使用 *step* 1, 它是穷尽的, 但却很慢。

结构 *Tac* (图10-13) 显示了策略可以如何简洁地表示证明过程。*safe* 声明列出了必需的策略, 通过 *firstF* 组合在一起。只产生一个子目标的策略被放在产生两个子目标的策略之前, 除此以外的顺序是任意的。通过策略算子 *--* 和 *repeatDeterm* 对 *safe* 进行重复, 就得到了 *safeSteps*。可以看出, *quant* 展开了至少一个量词, 也可能展开两个: 如果 *allL* 成功, 则进一步尝试 *exR*。

```

structure Tac : TAC =
  struct
    local open Tactical Rule
    in
      val safe =
        firstF [basic,
                conjL, disjR, impR, negL, negR, exL, allR, (* 一个子目标 *)
                conjR, disjL, impL, iffL, iffR             (* 两个子目标 *)];
      fun safeSteps i = safe i -- repeatDeterm (safe i);
      fun quant i     = (allL i -- try (exR i)) || exR i;
      val depth      = depthFirst final (safeSteps 1 || unify 1 || quant 1);
      fun step i     = safeSteps i || (unify i |@| allL i |@| exR i);
      fun depthIt d  = depthIter (final, d) (step 1);
    end
  end;
end;

```

图10-13 结构 *Tac*

在两个搜索策略之中, *depth* 要快一些, 但不完全是这样。它使用深度优先搜索, 这有进入死胡同的可能。另外, 它尽可能应用 *Rule.unify*, 而不理会其对于其他目标的影响。策略 *depthIt* 修正了这些缺点。注意, *step* 使用的是 *|@|* 而不是 *||* 来将 *Rule.unify* 与其他策略组合, 即使合一成功, 搜索也会查看量词展开。两种搜索都使用 *Rule.final* (图10-7) 来检测证明的结束状态。

让我们试着将 *Tac.depth* 用在 Pelletier (1986) 中的一些问题上。这是第39个问题:

```

goal "~ (EX x. ALL y. J(x,y) <-> ~J(y,y))";
> ~ (EX x. ALL y. J(x,y) <-> ~J(y,y))
> 1. empty |- ~ (EX x. ALL y. J(x,y) <-> ~J(y,y))

```

应用*Tac.depth*可以证明它:

```
by Tac.depth;
> ~(EX x. ALL y. J(x, y) <-> ~J(y, y))
> No subgoals left!
```

第40个问题更为复杂。^①

```
goal "(EX y. ALL x. J(y, x) <-> ~J(x, x)) --> \
\      ~ (ALL x. EX y. ALL z. J(z, y) <-> ~ J(z, x))";
> (EX y. ALL x. J(y, x) <-> ~J(x, x)) -->
> ~(ALL x. EX y. ALL z. J(z, y) <-> ~J(z, x))
> 1. empty
>   |- (EX y. ALL x. J(y, x) <-> ~J(x, x)) -->
>       ~(ALL x. EX y. ALL z. J(z, y) <-> ~J(z, x))
```

这个问题也很容易就证明了。

```
by Tac.depth;
> (EX y. ALL x. J(y, x) <-> ~J(x, x)) -->
> ~(ALL x. EX y. ALL z. J(z, y) <-> ~J(z, x))
> No subgoals left!
```

442

第42个问题更难一些: *Tac.depth*不能结束返回。

```
goal "~(EX y. ALL x. p(x, y) <-> ~(EX z. p(x, z) & p(z, x)))";
> ~(EX y. ALL x. p(x, y) <-> ~(EX z. p(x, z) & p(z, x)))
> 1. empty
>   |- ~(EX y.
>         ALL x.
>         p(x, y) <-> ~(EX z. p(x, z) & p(z, x)))
```

但是我们的另一个搜索策略成功了:

```
by (Tac.depthlt 1);
> ~(EX y. ALL x. p(x, y) <-> ~(EX z. p(x, z) & p(z, x)))
> No subgoals left!
```

需要重申的是, 我们的策略不能和自动定理证明机相比。这些策略是通过应用原始推理规则来工作的, 而这些规则的实现是针对交互式应用而设计的。这些策略的“内循环”(策略*safe*)以极其浪费的方式搜索连接词。量词的展开也缺少启发式的引导。下面这个看上去简单的例子(第43个问题)就不能在合理的时间内解决:

```
goal "(ALL x. ALL y. q(x, y) <-> (ALL z. p(z, x) <-> p(z, y))) \
\      --> (ALL x. (ALL y. q(x, y) <-> q(y, x)))";
```

策略最适用于当逻辑没有已知的自动证明步骤时。策略算子允许实验不同的搜索过程, 而抽象类型*state*杜绝了错误的论证。

① 其他定理证明机。大多数的自动定理证明机都是基于归结原理(Chang和Lee, 1973)的。它们证明公式*A*的方法是, 将 $\neg A$ 转换成子句形式(基于合取范式)并推导出一个矛盾。一个有名的归结证明机是W. McCune的Otter。Quaife (1992)曾经使

① 由于目标公式在一行内写不下, 转义序列\... \将字符串分写在两行。

用Otter进行Peano算术、几何和集合论中的证明。另一个令人印象深刻的系统是SETHEO (Letz等, 1992)。

表方法证明机功能较弱, 但是比归结证明机更为自然, 因为它们不需要将公式转换为子句形式。这方面的例子包括HARP (Oppacher和Suen, 1988) 以及出奇简单的leanTAP (Beckert和Posegga, 1995), 它只是由数行Prolog代码构成。策略depthIt有些基于leanTAP, 但却慢很多。

策略方案将适量的自动化和很大的灵活性结合起来。应用它的系统并不是针对古典的一阶逻辑, 而是其他具有计算重要性的逻辑。LCF支持论域理论的一种逻辑 (Gordon等, 1979; Paulson, 1987)。HOL系统支持丘奇的高阶逻辑 (Gordon和Melham, 1993)。Nuprl支持构造性类型理论的一种形式 (Constable等, 1986)。Isabelle是一种广泛的定理证明机, 支持多种不同的逻辑 (Paulson, 1994)。

443

练习 10.31 画图说明Hal中的结构、签名和函子, 以及它们之间的关系。

练习 10.32 为数学归纳规则实现一个策略, 其中包括常量0和后继函数suc:

$$\frac{\Gamma \vdash \Delta, \phi[0/x] \quad \phi, \Gamma \vdash \Delta, \phi[suc(x)/x]}{\Gamma \vdash \Delta, \forall x. \phi} \quad \text{限制条件: } x \text{ 不能在结论中自由出现}$$

你能预测将此策略加入到一个自动证明过程中会有什么困难吗?

练习 10.33 声明一个策略算子someGoal, 使得当其应用到具有n个子目标的状态上时, someGoal f等价于

$$f(n) \parallel f(n-1) \parallel \dots \parallel f(1)$$

repeat (someGoal Rule.conjR) 对一个证明状态做了些什么?

练习 10.34 我们的证明过程总是处理子目标1。何时选择其他子目标会更好?

要点小结

- 相继式演算是一阶逻辑的一个合适的证明系统。
- 合一辅助进行有关量词的论证。
- 合一中的出现检测对于正确性是至关重要的。
- 量化变量可以像λ-演算中的约束变量那样被处理。
- 推理规则可以作为定理或证明的抽象类型上的操作。
- 操作--、||和repeat在函数式程序设计的各个方面有着类似的功能。
- 策略方案允许自动化和交互式混合的定理证明。

444

项目建议

书中练习的用意在于加深对ML的理解，以及提高程序设计技能。然而，仅做这样的练习并不能成为一个程序员，更不用说是软件工程师了。项目比大型的练习涉及的内容更多，它涉及更多的程序设计。它需要认真地筹划：背景知识学习、需求分析和设计。完成后的程序还需要一定的评测，尽管不可能全面地评测。

这里的每个建议无异于一种提示，但是，稍加修改就可以得到一个真正的计划。首先，查阅后面给出的参考文献，给出项目描述，包括写出目的、预计时间安排，以及需要的资源列表。其次，是书写详细的需求分析，详细地列出所有功能，以便他人可以进行最终的测试。紧接着描述基本设计，ML的函子和签名可以刻画主要组件和它们的接口。

上面勾勒的准备阶段可以由导师、单个学生或一组学生完成。这取决于课程的目的，它有可能只是关心ML语言，或者是项目管理，或者是说明软件工程的某些方法。最后的评测也许同样可以由导师、实现者或者另一组学生来完成。

评测应该考虑程序和目标的吻合程度。测试可以由需求分析驱动。很多项目都不难完成，但是很难做到高效。所以评测除了考虑正确性之外，还有考虑性能，评测工具可以定位性能瓶颈。学生可能需要发掘和使用标准库的模块——数组、机器字操作等——这是取得高效的恰当途径。

这里建议的部分项目已经由剑桥的学生完成了，尽管不一定是用ML来完成的。其他的项目是因为有意思（至少对我来说）才列出来的，并且都有一定的难度。它们都特别适合用ML来做，实际上所有项目都可以用ML来完成，除了那些不需要安全性程序设计的或嵌入在使用其他语言的系统中的项目。所以，根据需要采用其他地方的项目建议。

无限精度的整数运算可以给出精确的答案，且不会溢出。（有些ML系统默认地提供这个功能。）Knuth（1981）讲述了这个算法，除了除法以外都很直接。他还提出了改进有理数运算的算法。

445

无限精度的实数运算可以产生满足任何给定精度的答案，自动地确定中间计算的精度。很多努力都用在寻找最高效的实数表示方法上（Boehm和Cartwright，1990）。Ménissier-Morain（1995）推荐了一种收敛的有理级数，它形如 p/B^q 。这种模式中的计算技巧是奇妙的，虽然十分困难（Gourdon和Salvy，1993）。你还可以拓展5.15节的数值例子。

第3章的多项式运算例子可以在几个方向上扩展。你可以提供更多的运算，允许多个变量的运算，或者还可以实现更好的GCD算法。见Davenport等（1993）或Knuth（1981）。无限精度的整数是必不可少的。

模拟器也很有趣：你可以使过时的机器和它曾运行过的软件获得新生。我个人的挚爱是DEC PDP-8。基本模式可以寻址4096个12位字，且带有一个含8个操作码的指令集。手册已经绝版了，但是资料在互联网上仍可以查到（Jones，1995）。诸如精确处理中断这样的细节是很棘手的。模拟运行的软件必须有一定的速度，至少要赶上用户的打字速度！

先进的重言式检测器包括有序二元决策图 (OBDD) 和 Davis-Putnam 证明过程。OBDD 在硬件和系统验证上都有应用, Bryant (1992) 是一个经典的描述, 而 Moore (1994) 可能更为适宜函数式程序设计。Davis-Putnam 在多年后重回人们的视线, 虽然早期的书本上有所介绍 (Chang 和 Lee, 1973), 但是, 最近的算法只在技术报告中有所讲述 (Zhang 和 Stickel, 1994)。

定理证明机可以有多种方式建立在第 10 章提供的基础上。表方法易于实现 (Beckert 和 Posegga, 1995)。模型消解 (model elimination) 也相当直接 (Stickel, 1988a)。Andrews (1989) 在有关高阶逻辑的背景下讲述了矩阵方法; 它也同样适合一阶逻辑。但是最有能力的学生应该尽力去实现消解法 (Stickel, 1988b), 为实现高性能而做的细化需要非常复杂的数据结构 (Butler 和 Overbeek, 1994)。

考虑书写一个语法分析器生成程序 (parser generator): 简单的 LR(0)、SLR 或一个 LALR(1) 版本, 并带有成熟的错误恢复功能。好的编译程序课本, 例如 Aho 等 (1986), 讲述了所需的技术。

编译项目总是很受欢迎的。选择一个小的 ML 子集, 并书写它的解释器。SECD 机器产生传值调用语义, 而图归约则是传需调用。Field 和 Harrison (1988) 讲述了实现方法, 以及类型检测。除非语法非常简单, 否则应该使用一个语法分析程序生成器, 例如 ML-Yacc (Tarditi 和 Appel, 1994)。

在尝试做一定规模的项目之前, 你应该熟读至少第 2 ~ 5 章, 若能连同第 7 章和第 8 章一起看, 就更好了。祝好运!

参考文献

- Aasa, A., Holmström, S., and Nilsson, C. (1988). An efficiency comparison of some representations of purely functional arrays. *BIT*, **28**, 490–503.
- Abelson, H. and Sussman, G. J. (1985). *Structure and Interpretation of Computer Programs*. MIT Press.
- Adams, S. (1993). Efficient sets – a balancing act. *Journal of Functional Programming*, **3**(4), 553–561.
- Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques and Tools*. Addison-Wesley.
- Andrews, P. B. (1989). On connections and higher-order logic. *Journal of Automated Reasoning*, **5**(3), 257–291.
- Appel, A. W. (1992). *Compiling with Continuations*. Cambridge University Press.
- Appel, A. W. (1993). A critique of standard ML. *Journal of Functional Programming*, **3**(4), 391–429.
- Appel, A. W., Mattson, J. S., and Tarditi, D. R. (1994). *A Lexical Analyzer Generator for Standard ML*, version 1.5.0 edition. Distributed with Standard ML of New Jersey and through freeware archives.
- Augustsson, L. and Johnsson, T. (1989). The Chalmers Lazy-ML compiler. *Computer Journal*, **32**, 127–141.
- Backus, J. (1978). Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, **21**, 613–641.
- Bailey, R. (1990). *Functional Programming with Hope*. Ellis Horwood.
- Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics*. North-Holland.
- Beckert, B. and Posegga, J. (1995). leanTAP: Lean tableau-based deduction. *Journal of Automated Reasoning*, **15**(3), 339–358.
- Bevier, W. R., Hunt, Jr., W. A., Moore, J. S., and Young, W. D. (1989). An approach to systems verification. *Journal of Automated Reasoning*, **5**(4), 411–428.
- Biagioni, E., Harper, R., Lee, P., and Milnes, B. G. (1994). Signatures for a network protocol stack: A systems application of Standard ML. In *LISP and Functional Programming*, pages 55–64.
- Bird, R. and Wadler, P. (1988). *Introduction to Functional Programming*. Prentice-Hall.
- Boehm, H. and Cartwright, R. (1990). Exact real arithmetic: Formulating real

- numbers as functions. In Turner (1990b), pages 43–64.
- Boolos, G. S. and Jeffrey, R. C. (1980). *Computability and Logic*. Cambridge University Press, 2nd edition.
- Boyer, R. S. and Moore, J. S. (1988). *A Computational Logic Handbook*. Academic Press.
- Bryant, R. E. (1992). Symbolic boolean manipulation with ordered binary-decision diagrams. *Computing Surveys*, **24**(3), 293–318.
- Burge, W. H. (1975). *Recursive Programming Techniques*. Addison-Wesley.
- Burton, F. W. (1982). An efficient functional implementation of FIFO queues. *Information Processing Letters*, **14**, 205–206.
- Butler, R. M. and Overbeek, R. A. (1994). Formula databases for high-performance resolution/paramodulation systems. *Journal of Automated Reasoning*, **12**(2), 139–156.
- Cann, D. (1992). Retire Fortran? a debate rekindled. *Communications of the ACM*, **35**(8), 81–89.
- Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, **17**, 471–522.
- Chang, C.-L. and Lee, R. C.-T. (1973). *Symbolic Logic and Mechanical Theorem Proving*. Academic Press.
- Cohn, A. (1989a). Correctness properties of the Viper block model: The second level. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 1–91. Springer.
- Cohn, A. (1989b). The notion of proof in hardware verification. *Journal of Automated Reasoning*, **5**(2), 127–139.
- Constable, R. L. et al. (1986). *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall.
- Corbin, J. and Bidoit, M. (1983). A rehabilitation of Robinson’s unification algorithm. In R. E. A. Mason, editor, *Information Processing 83*. IFIP, Elsevier.
- Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to Algorithms*. MIT Press.
- Cousineau, G. and Huet, G. (1990). The CAML primer. Technical report, INRIA, Rocquencourt, France.
- Damas, L. and Milner, R. (1982). Principal type-schemes for functional programs. In *9th Symposium on Principles of Programming Languages*, pages 207–212. ACM.
- Davenport, H. (1952). *The Higher Arithmetic*. Cambridge University Press, 6th edition.
- Davenport, J. H., Siret, Y., and Tournier, E. (1993). *Computer Algebra: Systems and Algorithms for Algebraic Computation*. Academic Press.
- de Bruijn, N. G. (1972). Lambda calculus notation with nameless dummies, a

- tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indagationes Mathematicae*, **34**, 381–392.
- Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall.
- Feynman, R. P., Leighton, R. B., and Sands, M. (1963). *The Feynman Lectures on Physics*, volume 1. Addison-Wesley.
- Field, A. J. and Harrison, P. G. (1988). *Functional Programming*. Addison-Wesley.
- Fitzgerald, J. S., Larsen, P. G., Brookes, T. M., and Green, M. A. (1995). Developing a security-critical system using formal and conventional methods. In Hinchey and Bowen (1995), pages 333–356.
- FPCA (1995). *Functional Programming and Computer Architecture*. ACM Press.
- Frost, R. and Launchbury, J. (1989). Constructing natural language interpreters in a lazy functional language. *Computer Journal*, **32**, 108–121.
- Gallier, J. H. (1986). *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row.
- Galton, A. (1990). *Logic for Information Technology*. Wiley.
- Gansner, E. R. and Reppy, J. H., editors (1996). *The Standard ML Basis Library Reference Manual*. In preparation.
- Gerhart, S., Craigen, D., and Ralston, T. (1994). Experience with formal methods in critical systems. *IEEE Software*, pages 21–28.
- Gordon, M. J. C. (1988). *Programming Language Theory and its Implementation*. Prentice-Hall.
- Gordon, M. J. C. and Melham, T. F. (1993). *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press.
- Gordon, M. J. C., Milner, R., and Wadsworth, C. P. (1979). *Edinburgh LCF: A Mechanised Logic of Computation*. Springer. LNCS 78.
- Gourdon, X. and Salvy, B. (1993). Computing one million digits of $\sqrt{2}$. Technical Report 155, INRIA-Rocquencourt.
- Graham, B. T. (1992). *The SECD Microprocessor: A Verification Case Study*. Kluwer Academic Publishers.
- Grant, P. W., Sharp, J. A., Webster, M. F., and Zhang, X. (1995). Experiences of parallelising finite-element problems in a functional style. *Software—Practice and Experience*, **25**(9), 947–974.
- Grant, P. W., Sharp, J. A., Webster, M. F., and Zhang, X. (1996). Sparse matrix representations in a functional language. *Journal of Functional Programming*, **6**(1), 143–170.
- Greiner, J. (1996). Weak polymorphism can be sound. *Journal of Functional Programming*, **6**(1), 111–141.
- Gunter, C. A. (1992). *Semantics of Programming Languages: Structures and Techniques*. MIT Press.
- Halfant, M. and Sussman, G. J. (1988). Abstraction in numerical methods. In *LISP and Functional Programming*, pages 1–7. ACM Press.

- Harper, R. (1994). A simplified account of polymorphic references. *Information Processing Letters*, 51(4), 201–206.
- Harper, R., MacQueen, D., and Milner, R. (1986). Standard ML. Technical Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh.
- Hartel, P. and Plasmeijer, R. (1996). State-of-the-art applications of pure functional programming. *Journal of Functional Programming*, 5(3). Special issue.
- Hennessy, M. (1990). *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. Wiley.
- Hinchey, M. and Bowen, J. P., editors (1995). *Applications of Formal Methods*. Prentice-Hall.
- Hoare, C. A. R. (1989a). Computer science. In Hoare and Jones (1989), pages 89–101. Inaugural lecture, Queen's University of Belfast, 1971.
- Hoare, C. A. R. (1989b). Hints on programming-language design. In Hoare and Jones (1989), pages 193–216. First appeared in 1974.
- Hoare, C. A. R. (1989c). An overview of some formal methods for program design. In Hoare and Jones (1989), pages 371–387. Reprinted from *Computer* 20 (1987), 85–91.
- Hoare, C. A. R. and Jones, C. B., editors (1989). *Essays in Computing Science*. Prentice-Hall.
- Hoogerwoord, R. (1992). A logarithmic implementation of flexible arrays. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction: Second International Conference*, LNCS 669, pages 191–207. Springer.
- Hudak, P., Jones, S. P., and Wadler, P. (1992). Report on the programming language Haskell: A non-strict, purely functional language. *SIGPLAN Notices*, 27(5). Version 1.2.
- Huet, G. P. (1975). A unification algorithm for typed λ -calculus. *Theoretical Computer Science*, 1, 27–57.
- Hughes, J. (1989). Why functional programming matters. *Computer Journal*, 32, 98–107.
- Jones, D. W. (1995). The Digital Equipment Corporation PDP-8. On the World Wide Web at URL <http://www.cs.uiowa.edu/~jones/pdp8/>.
- Kennedy, A. (1996). Functional pearls: Drawing trees. *Journal of Functional Programming*, 6(3), 527–534.
- Knuth, D. E. (1973). *The Art of Computer Programming*, volume 3: *Sorting and Searching*. Addison-Wesley.
- Knuth, D. E. (1981). *The Art of Computer Programming*, volume 2: *Seminumerical Algorithms*. Addison-Wesley, 2nd edition.
- Korf, R. E. (1985). Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27, 97–109.
- Lakatos, I. (1976). *Proofs and Refutations: The Logic of Mathematical*

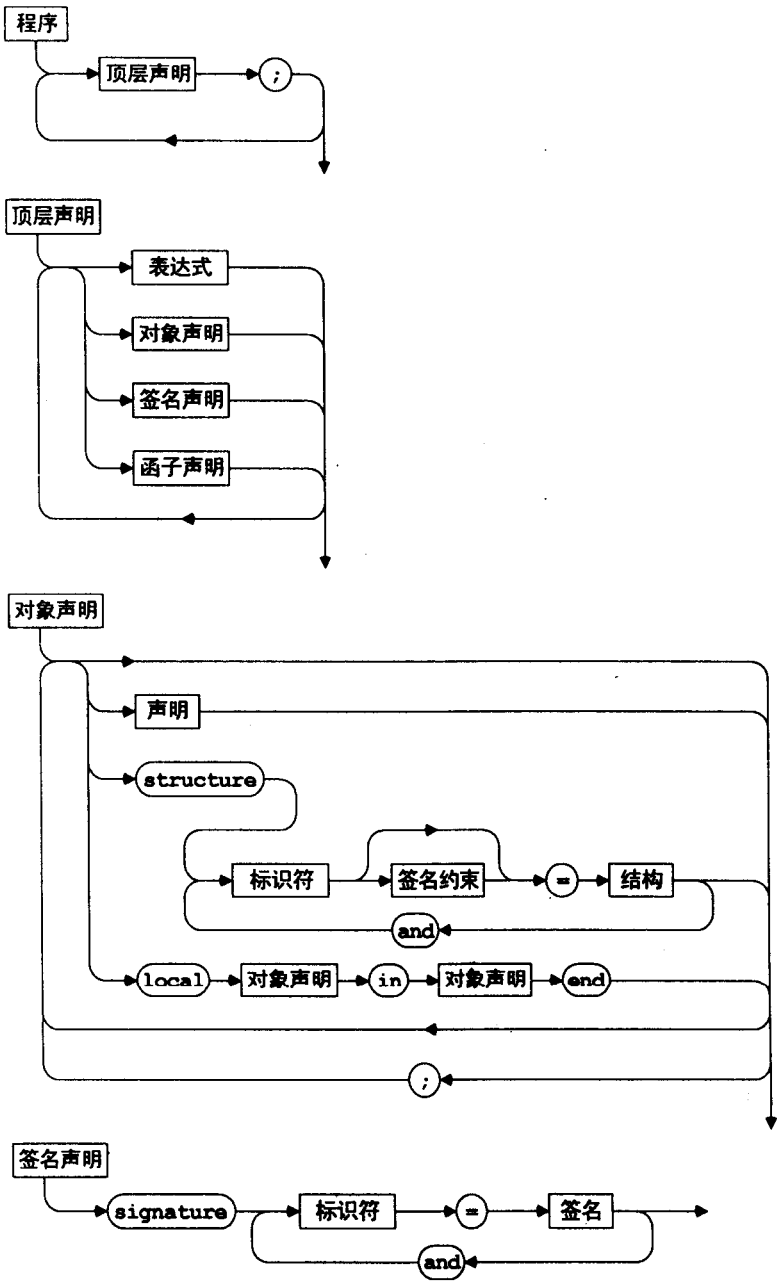
- Discovery*. Cambridge University Press.
- Landin, P. J. (1966). The next 700 programming languages. *Communications of the ACM*, 9(3), 157–166.
- Leroy, X. and Mauny, M. (1993). Dynamics in ML. *Journal of Functional Programming*, 3(4), 431–463.
- Letz, R., Schumann, J., Bayerl, S., and Bibel, W. (1992). SETHEO: A high-performance theorem prover. *Journal of Automated Reasoning*, 8(2), 183–212.
- Magnusson, L. and Nordström, B. (published 1994). The ALF proof editor and its proof engine. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs: International Workshop TYPES '93*, LNCS 806, pages 213–237. Springer.
- Martelli, A. and Montanari, U. (1982). An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4, 258–282.
- Mattson, Jr., H. F. (1993). *Discrete Mathematics with Applications*. Wiley.
- McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., and Levin, M. I. (1962). *LISP 1.5 Programmer's Manual*. MIT Press.
- Ménissier-Morain, V. (1995). Arbitrary precision real arithmetic: design and algorithms. Submitted to *Journal of Symbolic Computation*.
- Mills, H. D. and Linger, R. C. (1986). Data structured programming: Program design without arrays and pointers. *IEEE Transactions on Software Engineering*, SE-12, 192–197.
- Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 348–375.
- Milner, R. and Tofte, M. (1990). *Commentary on Standard ML*. MIT Press.
- Milner, R., Tofte, M., and Harper, R. (1990). *The Definition of Standard ML*. MIT Press.
- Moore, J. S. (1994). Introduction to the OBDD algorithm for the ATP community. *Journal of Automated Reasoning*, 12(1), 33–46.
- Nederpelt, R. P., Geuvers, J. H., and de Vrijer, R. C., editors (1994). *Selected Papers on Automath*. North-Holland.
- Odersky, M., Wadler, P., and Wehr, M. (1995). A second look at overloading. In FPCA (1995), pages 135–146.
- Ohuri, A. (1995). A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6), 844–895.
- Okasaki, C. (1995). Purely functional random-access lists. In FPCA (1995), pages 86–95.
- O'Keefe, R. A. (1982). A smooth applicative merge sort. Research paper 182, Department of AI, University of Edinburgh.
- Oppacher, F. and Suen, E. (1988). HARP: A tableau-based theorem prover. *Journal of Automated Reasoning*, 4(1), 69–100.
- Oppen, D. C. (1980). Pretty printing. *ACM Transactions on Programming Languages and Systems*, 2, 465–483.

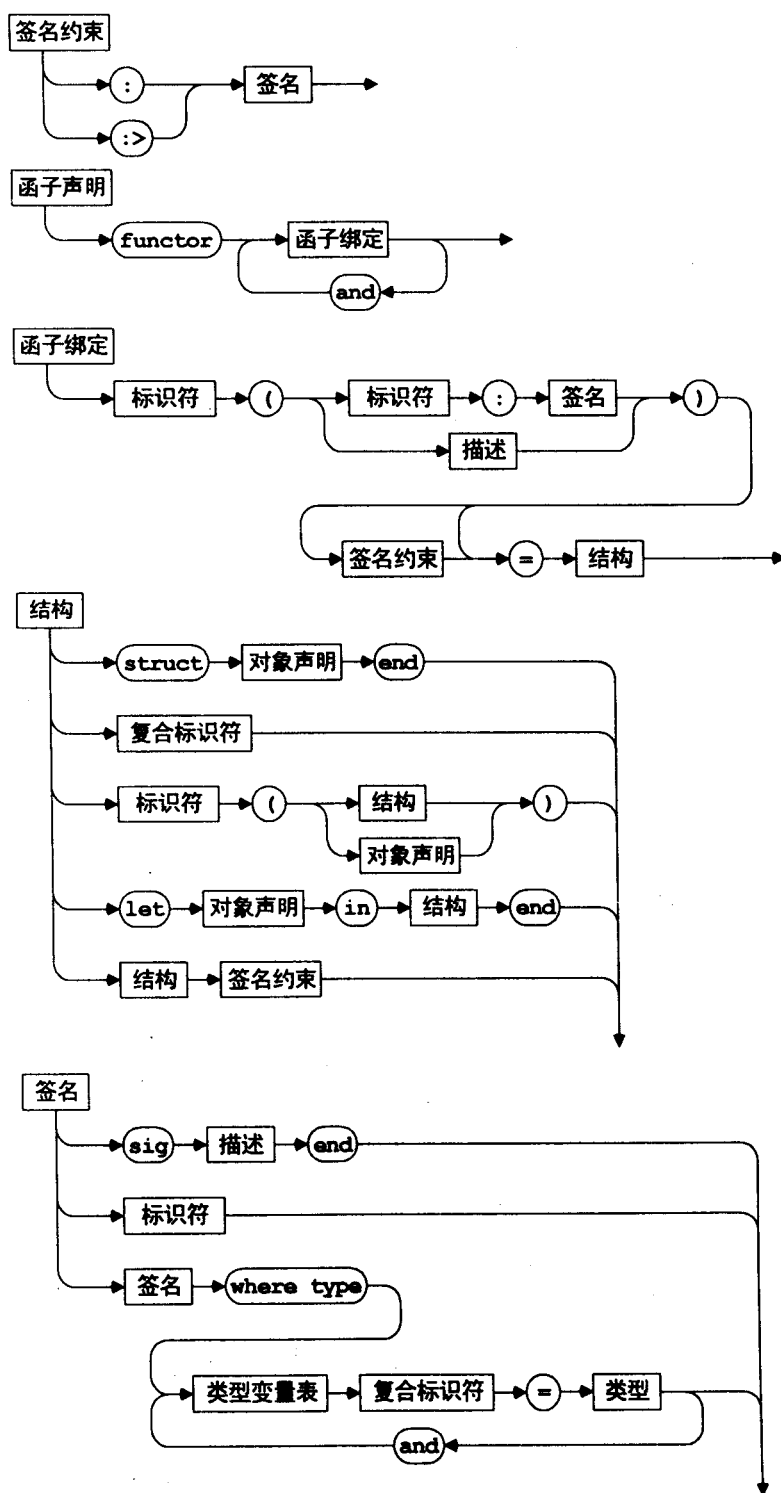
- Park, S. K. and Miller, K. W. (1988). Random number generators: Good ones are hard to find. *Communications of the ACM*, **31**, 1192–1201.
- Paterson, M. S. and Wegman, M. N. (1978). Linear unification. *Journal of Computer and System Sciences*, **16**, 158–167.
- Paulson, L. C. (1983). A higher-order implementation of rewriting. *Science of Computer Programming*, **3**, 119–149.
- Paulson, L. C. (1987). *Logic and Computation: Interactive proof with Cambridge LCF*. Cambridge University Press.
- Paulson, L. C. (1994). *Isabelle: A Generic Theorem Prover*. Springer. LNCS 828.
- Paulson, L. C. (1995). Set theory for verification: II. Induction and recursion. *Journal of Automated Reasoning*, **15**(2), 167–215.
- Pelletier, F. J. (1986). Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, **2**, 191–216. Errata, JAR 4 (1988), 235–236.
- Penrose, R. (1989). *The Emperor's New Mind: Concerning Computers, Minds, and the Laws of Physics*. Oxford University Press.
- Peyton Jones, S. L. (1992). Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, **2**(2), 127–202.
- Peyton Jones, S. L. and Wadler, P. (1993). Imperative functional programming. In *20th Symposium on Principles of Programming Languages*, pages 71–84. ACM Press.
- Quaife, A. (1992). Automated deduction in von Neumann-Bernays-Gödel set theory. *Journal of Automated Reasoning*, **8**(1), 91–147.
- Reade, C. (1989). *Elements of Functional Programming*. Addison-Wesley.
- Reade, C. (1992). Balanced trees with removals: an exercise in rewriting and proof. *Science of Computer Programming*, **18**, 181–204.
- Reeves, S. and Clarke, M. (1990). *Logic for Computer Science*. Addison-Wesley.
- Rich, E. and Knight, K. (1991). *Artificial Intelligence*. McGraw-Hill, 2nd edition.
- Ružička, P. and Prívara, I. (1988). An almost linear Robinson unification algorithm. In M. P. Chytil, L. Janiga, and V. Koubek, editors, *Mathematical Foundations of Computer Science*, pages 501–511. Springer. LNCS 324.
- Sedgewick, R. (1988). *Algorithms*. Addison-Wesley, 2nd edition.
- Sleator, D. D. and Tarjan, R. E. (1985). Self-adjusting binary search trees. *Journal of the ACM*, **32**(3), 652–686.
- Smith, M. H., Garigiliano, R., Morgan, R., Shiu, S., and Jarvis, S. (1994). LOLITA : A natural language engineered system. Technical report, Department of Computer Science, University of Durham.
- Spafford, E. H. (1989). The Internet worm: Crisis and aftermath.

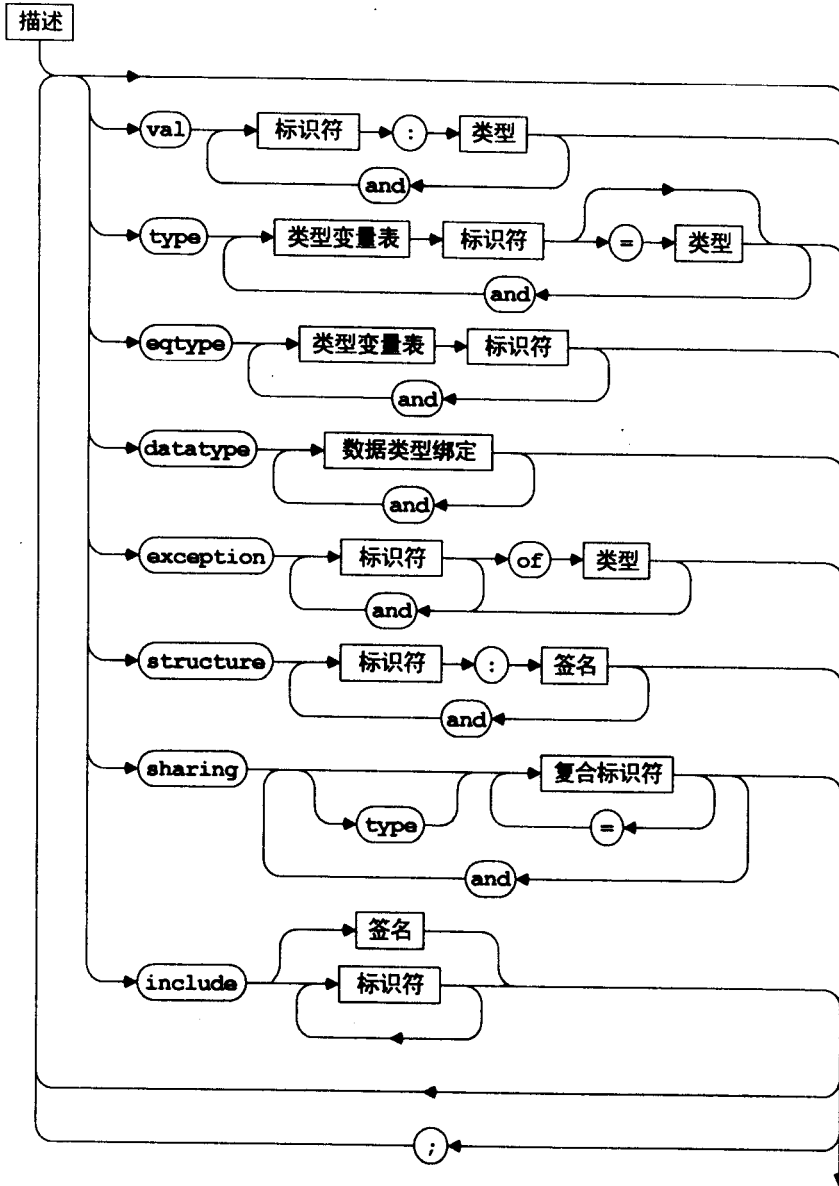
- Communications of the ACM*, 32(6), 678–687.
- Srivas, M. K. and Miller, S. P. (1995). Formal verification of the AAMP5 microprocessor. In Hinchey and Bowen (1995), pages 125–180.
- Stickel, M. E. (1988a). A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4(4), 353–380.
- Stickel, M. E. (1988b). Resolution theorem proving. *Annual Review of Computer Science*, 3, 285–316.
- Tarditi, D. R. and Appel, A. W. (1994). *ML-Yacc User's Manual*, version 2.2 edition. Distributed with Standard ML of New Jersey and through freeware archives.
- Thompson, S. (1991). *Type Theory and Functional Programming*. Addison-Wesley.
- Tofte, M. (1990). Type inference for polymorphic references. *Information and Computation*, 89, 1–34.
- Turner, D. (1990a). An overview of Miranda. In Turner (1990b), pages 1–16.
- Turner, D., editor (1990b). *Research Topics in Functional Programming*. Addison-Wesley.
- Turner, D. A. (1979). A new implementation technique for applicative languages. *Software—Practice and Experience*, 9, 31–49.
- Turner, R. (1991). *Constructive Foundations for Functional Languages*. McGraw-Hill.
- Uribe, T. E. and Stickel, M. E. (1994). Ordered binary decision diagrams and the Davis-Putnam procedure. In J. P. Jouannaud, editor, *Constraints in Computational Logics: First International Conference*, LNCS 845, pages 34–49. Springer.
- Wadler, P. and Gill, A. (1995). Real world applications of functional programming. On the World Wide Web at URL <http://www.dcs.gla.ac.uk/fp/realworld/>.
- Wiener, L. R. (1993). *Digital Woes: Why We Should Not Depend on Software*. Addison-Wesley.
- Winskel, G. (1993). *The Formal Semantics of Programming Languages*. MIT Press.
- Wirth, N. (1985). *Programming in Modula-2*. Springer, 3rd edition.
- Wright, A. K. (1995). Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4), 343–356.
- Zhang, H. and Stickel, M. E. (1994). An efficient algorithm for unit propagation. Technical Report 94-12, Computer Science Dept., University of Iowa.

Standard ML语法图

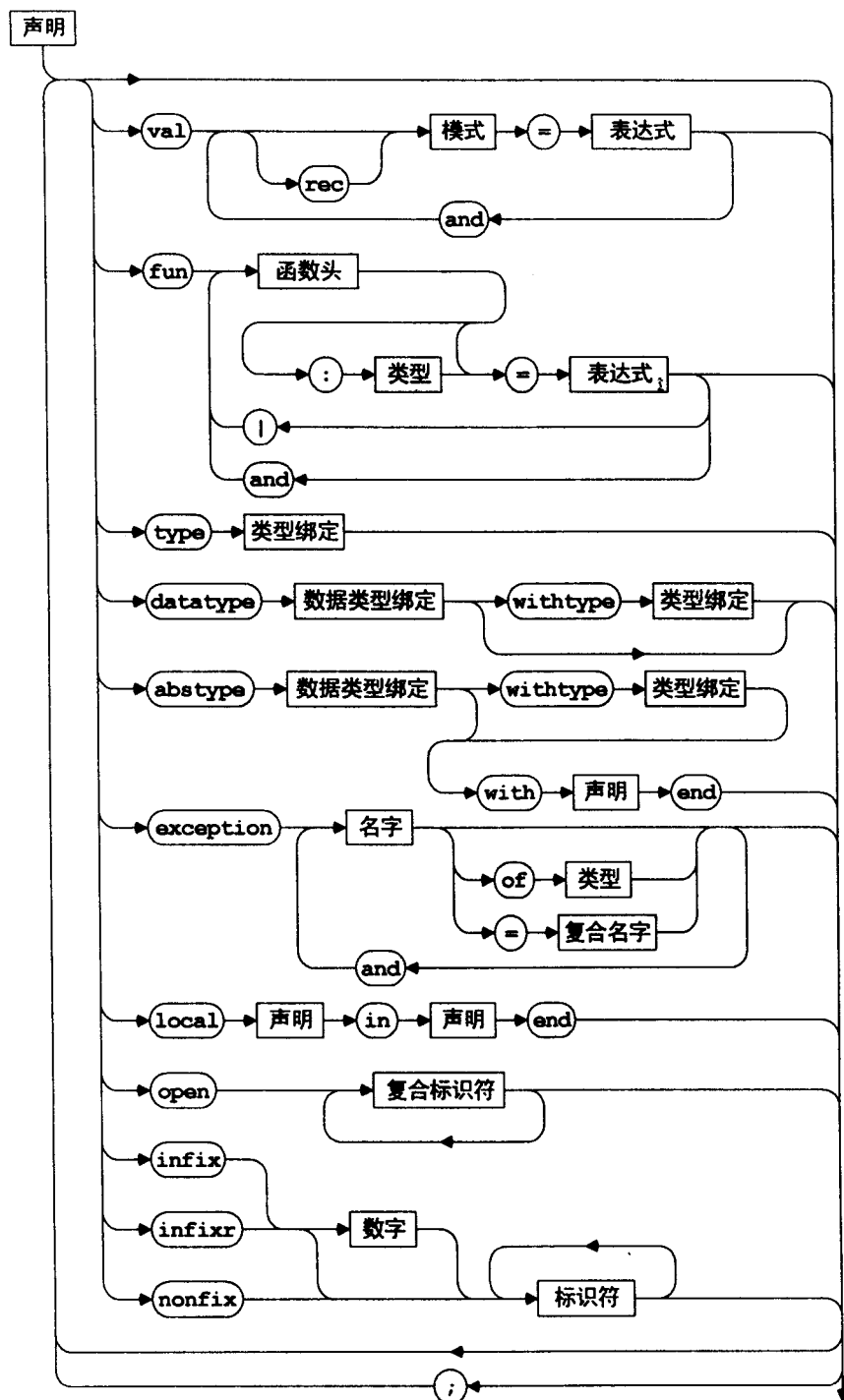
程序和模块

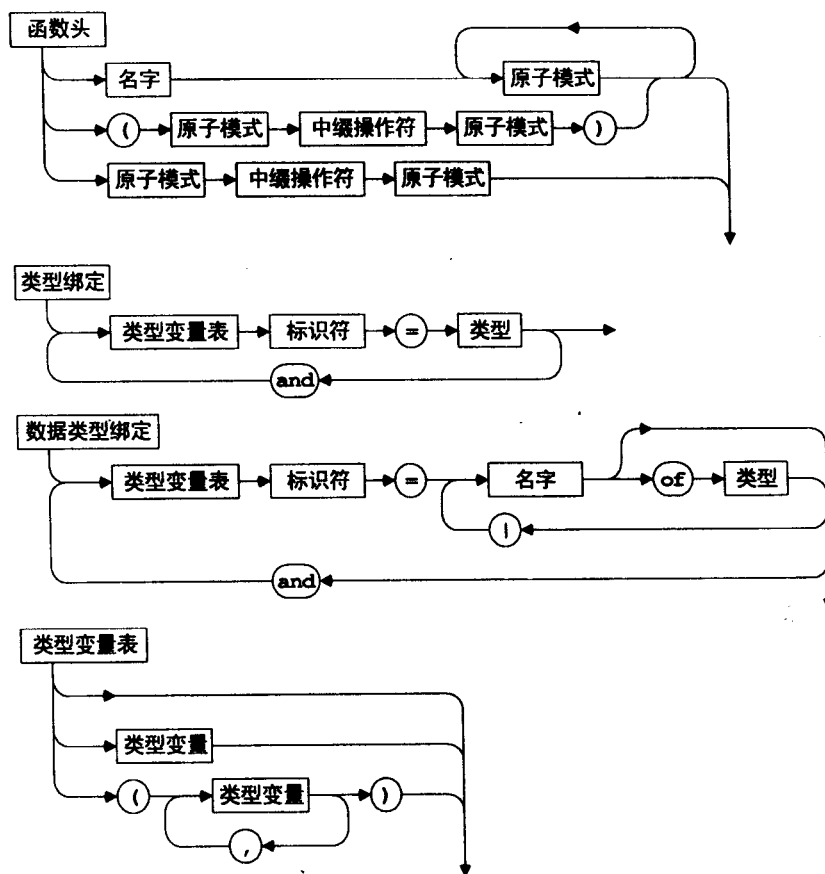




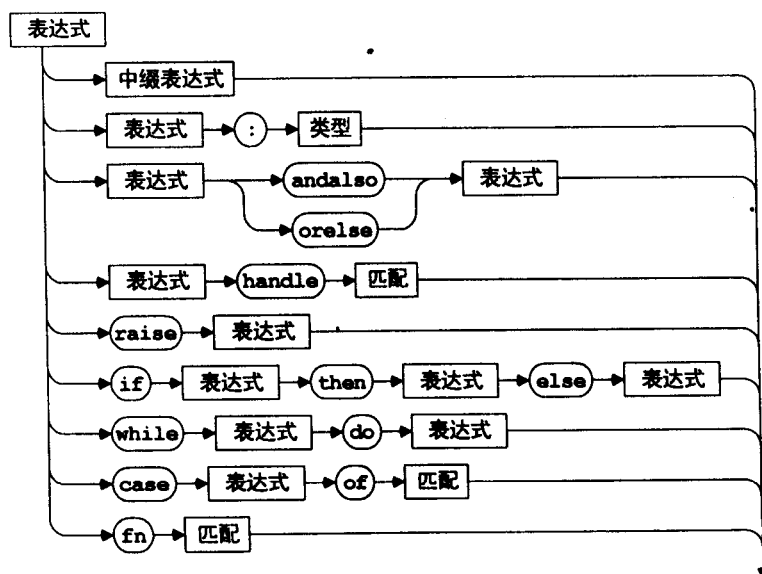


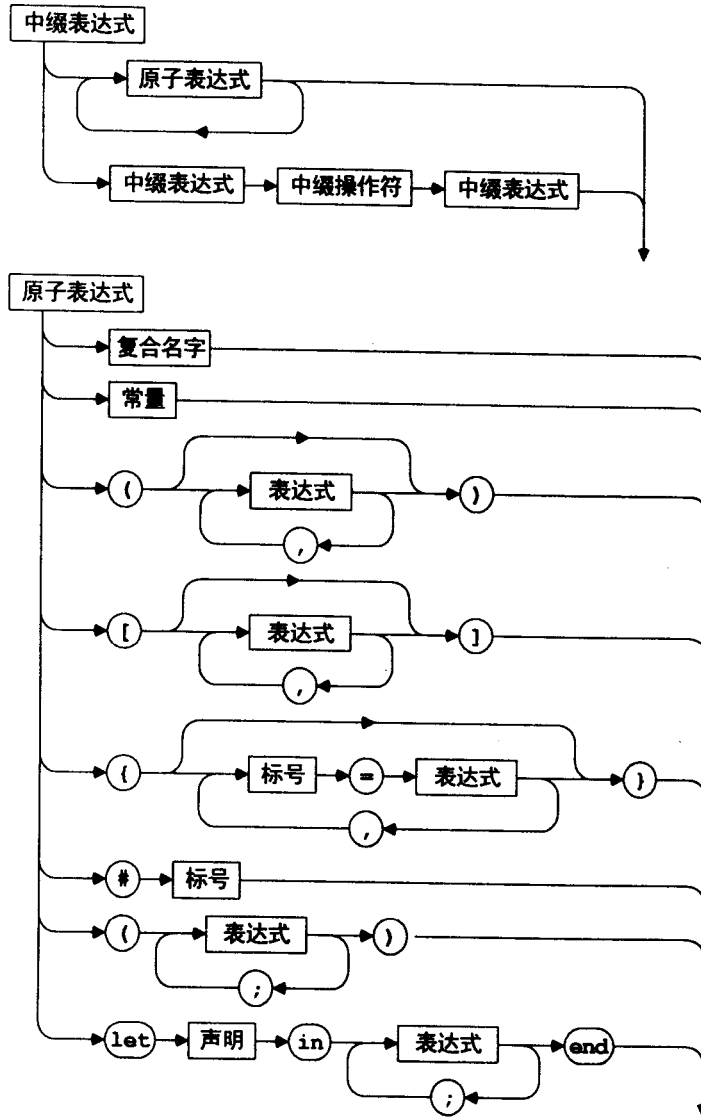
声明



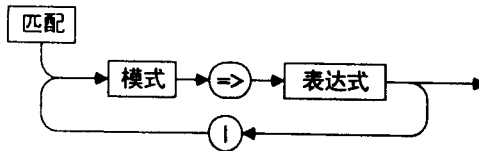


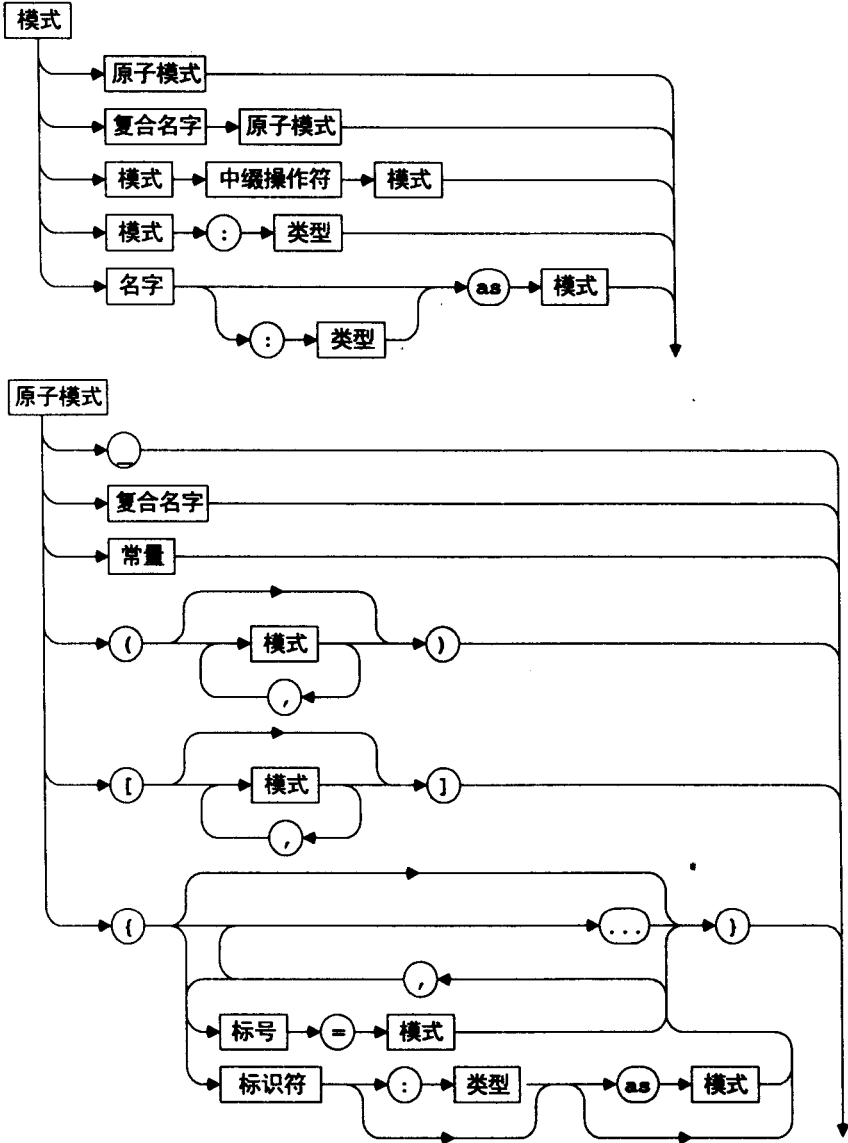
表达式



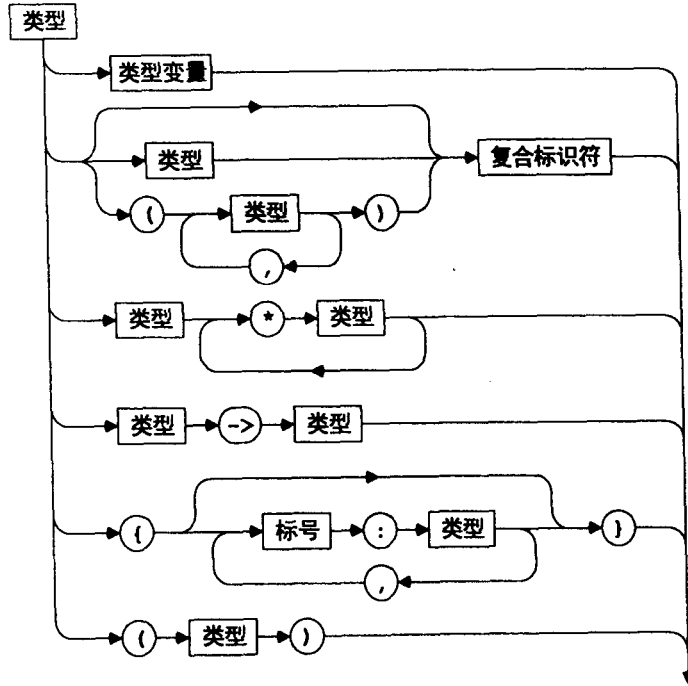


模式和匹配

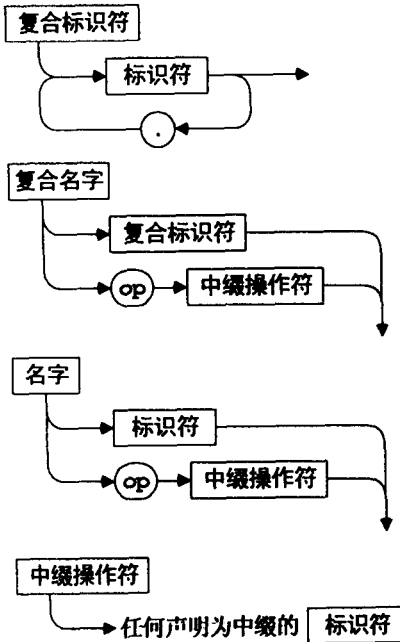


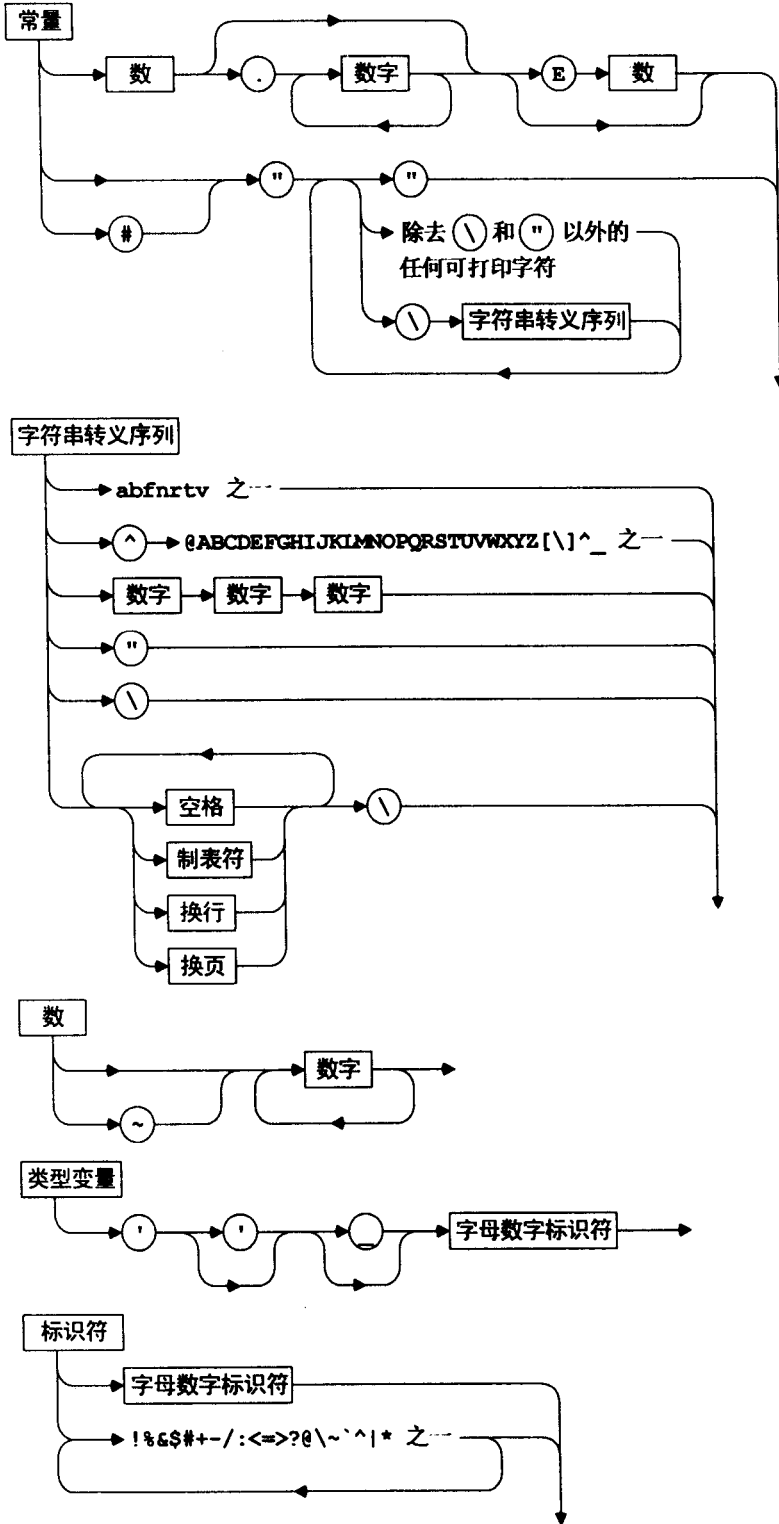


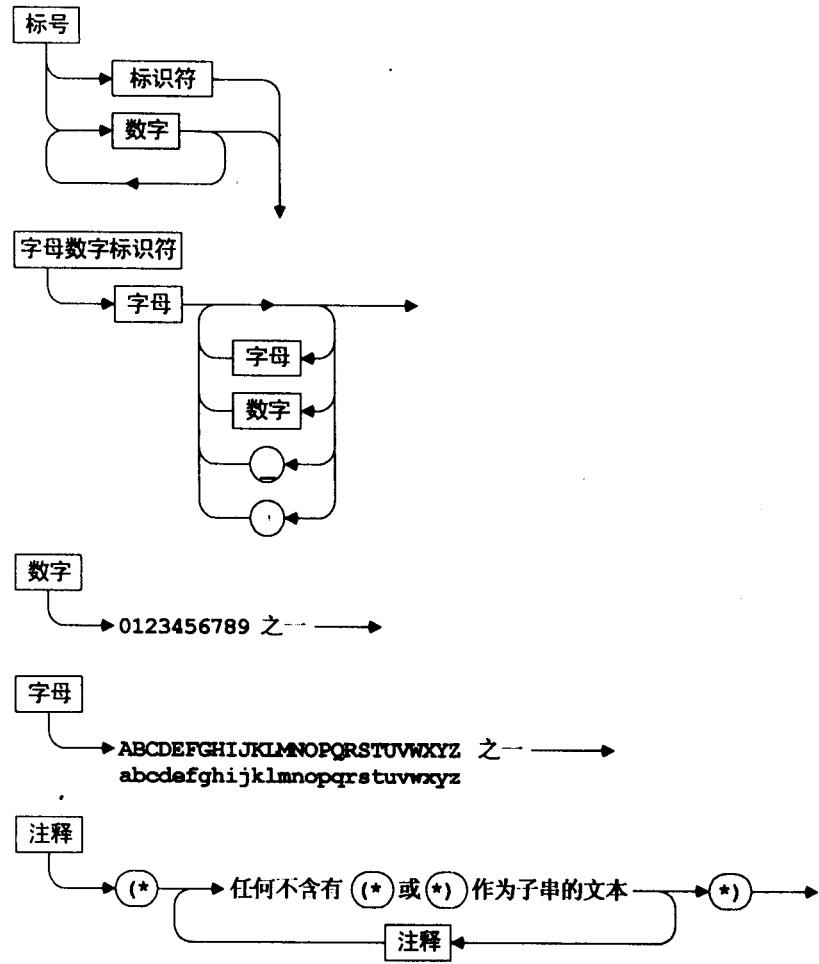
类型



词法相关：标识符、常量、注释







语法图中英词汇对照表

程序	Program	原子模式	Atomic Pattern
顶层声明	Top level Declaration	类型	Type
对象声明	Object Declaration	复合标识符	Compound Ident
签名声明	Signature Declaration	复合名字	Compound Name
签名约束	Sig Constraint	名字	Name
函子声明	Functor Declaration	中缀操作符	Infix Operator
函子绑定	Functor Binding	常量	Constant
结构	Structure	字符串转义序列	String Escape
签名	Signature	数	Numeral
描述	Specification	类型变量	Type Var
声明	Declaration	标识符	Ident
函数头	Function Heading	标号	Label
类型绑定	Type Binding	字母数字标识符	Alphanumeric Ident
数据类型绑定	Datatype Binding	数字	Digit
类型变量表	Type Var List	字母	Letter
表达式	Expression	注释	Comment
中缀表达式	Infix Expression	空格	Space
原子表达式	Atomic Expression	制表符	Tab
匹配	Match	换行	Newline
模式	Pattern	换页	Formfeed

索引

索引中的页码为英文原书页码, 与书中边栏页码一致。

! function(函数), 314-318
() constructor (构造子), 32
* infix (中缀), 22, 23, 303
+ infix (中缀), 22, 23, 303
- infix (中缀), 22, 23, 303
/ infix (中缀), 23, 303
:: constructor (构造子), 70-71, 77, 186
:= infix (中缀), 314-317
< infix (中缀), 26
<= infix (中缀), 27
<> infix (中缀), 27, 又见 equality (相等)
= infix (中缀), 27, 97, 又见 equality (相等)
=> keyword (关键字), 133, 138, 172, 463
> infix (中缀), 27
>= infix (中缀), 27
@ infix (中缀), 78-80, 82, 186
 eliminating (消除), 80, 111, 146
 for sequences (序列上的), 195
 proofs about (相关的证明), 226-229
^ infix (中缀), 25, 125
~ function (函数), 23

A

AAMP5 microprocessor (AAMP5 微处理器), 256
Aasa, Annika, 339
abs 函数, 24
abstract types (抽象类型), 97, 115, 263-269
 examples of (实例), 281-283, 327-334
 for proof systems (证明系统上的), 421
 how to declare (如何声明), 268
abstraction over variables (基于变量的抽象)
 in λ -calculus (λ -演算中的), 372-374, 377, 379
 in logic (逻辑中的), 407-409
abstype declarations (声明), 266-269, 281, 284, 460
 repeated (重复), 293
Adams, Stephen, 154
ALF system (ALF系统), 13
all function (函数), 184-185, 187
amortized cost (分摊的开销), 262
andalso keyword (关键字), 27, 43, 462
Andrews, Peter, 446
app function (函数), 320
Appel, Andrew, 10, 15, 102, 351, 371

append (追加, 连接), 见 @
applicative programming (应用式程序设计), 见 functional programming (函数式程序设计)
ARITH signature (签名), 62-63, 86, 116
arithmetic in ML (ML中的算术), 14, 22-24, 137, 465
 unlimited precision (无限精度), 445-446
Array structure (结构), 335
arrays (数组)
 flexible (弹性), 154-159, 263, 300
 functional (函数式), 336-339
 mutable (可变更), 154, 335-336
as keyword (关键字), 132, 133, 156, 463-464
 assignment commands (赋值命令), 见 :=
 assignments (赋值, 指派), in logic (逻辑中的), 398
 association lists (关联表), 101-102, 150, 287-288, 336
atan function (函数), 24
AUTOMATH system (AUTOMATH系统), 395

B

backtracking (回溯), 见 search (搜索), depth-first (深度优先)
Backus, John, 6注, 9
Beckert, B., 443
before infix (中缀), 320
Bevier, William R., 256
Biagioni, Edoardo, 10
binary arithmetic (二元算术运算), 85-87
Bind exception (异常), 137, 138
BinIO structure (结构), 350
Boehm, Hans, 446
Bool structure (结构), 340
bool type (类型), 26, 127
boolean values (布尔值), 26-27
 in λ -calculus (λ -演算中的), 385
Boyer/Moore theorem prover (Boyer/Moore定理证明机),
 见 NQTHM
Braun, W., 155
Bruijn, N. G. de, 395
Burge, W. H., 371

C

C (C语言), 15-16, 274
call-by-name (传名调用), 44-45, 194, 200注
in λ -calculus (λ -演算中的), 375, 388-392, 395

call-by-need (传需调用), 8, 9, 45-48, 140, 又见 sequences (序列)
 call-by-value (传值调用), 39-40, 43, 44, 136, 140
 in λ -calculus (λ -演算中的), 375, 389-393
 CAML, 12, 136
 Cardelli, Luca, 67
 Cartwright, Robert, 446
 case expressions (表达式), 133, 137-140, 173, 462
 ceil function (函数), 24
 Char structure (结构), 26, 341-342
 char type (类型), 25-26
 Chr exception (异常), 137
 chr function (函数), 25-26, 137
 Church, Alonzo (丘奇), 47, 372
 Church-Rosser Theorem (Church-Rosser定理), 374
 Cohn, Avra, 256
 combinators (组合子), 180-182
 comments (注释), 20, 467
 complex numbers (复数), 59-61
 composition, of functions (函数的复合), 见 *o*
 computer algebra (计算机代数), 114
 concat function (函数), 74, 82
 for lists (表上的), 81, 187, 190
 for sequences (序列上的), 437
 concatenation (连接)
 of lists (表的), 见 @
 in λ -calculus (λ -演算中的), 388, 395
 of sequences (序列的), 195
 of strings (字符串的), 25
 conditional expressions (条件表达式), 26-27, 43-44, 462
 and exceptions (及 异常), 137, 140
 in λ -calculus (λ -演算中的), 385, 391
 type checking of (之 类型检测), 64
 conjunctive normal form (合取范式), 167-170, 240-242
 cons (构造), 见 :: constructor (构造子)
 Constable, Robert, 443
 constructive type theory (构造性类型理论), 13, 443
 constructors (构造子), 125, 130-132
 for lists (表上的), 70
 hiding (隐藏), 159, 265-269
 control structures (控制结构), 317-321
 cos function (函数), 24
 Cousineau, Guy, 12

D

Damas, Luis, 67
 datatype bindings (数据类型绑定), 267, 461
 datatype declarations (声明), 124-130, 460
 recursive (递归的), 142, 165, 192, 194, 233
 repeated (重复的), 128, 293
 with one constructor (仅有一个构造子的), 159, 261

datatype specifications (描述), 310
 Date structure (结构), 15
 Davis-Putnam procedure (Davis-Putnam过程), 170, 446
 declarations (声明), 18-22, 53-56, 460, 又见每一种声明
 in a structure (结构中的), 60
 of modules (模块的), 311-312, 457-458
 simultaneous (同时的, 联立), 56-58
 declarative programming (声明式程序设计), 10
 depth function (函数), 143, 189, 232
 dereferencing (引用解析), 见 !
 Dictionary functor (函子), 281-283
 DICTIONARY signature (签名), 149-150, 266, 281, 288
 Dijkstra, Edsger (迪克斯特拉), 82, 93, 94, 159
 disjoint sum type (不相交和类型), 129-130
 disjunctive normal form (析取范式), 170
 distrib function (函数), 见 conjunctive normal form (合取范式)
 Div exception (异常), 137
 div infix (中缀), 22, 49, 64
 Domain exception (异常), 137
 domain theory (论域理论), 215, 216, 233, 247, 443
 drop function (函数), 78, 82, 111, 188, 424
 proofs about (相关的证明), 251-254
 dropwhile function (函数), 184

E

efficiency (效率), 9-10, 47
 of recursion (递归的), 42, 76-80
 Eight Queens problem (八皇后问题), 208-211
 Empty exception (异常), 138
 environments (环境), 21, 39, 62, 378, 393, 418
 eqtype specifications (描述), 266, 269, 287-288, 310, 459
 EQUAL constructor (构造子), 127, 281, 289
 equality (相等), 96-107
 and abstract types (及 抽象类型), 97, 267, 268
 and functions (及 函数), 97, 234-236
 of references (引用的), 316, 332-334
 Euclid's Algorithm (欧几里得算法, 辗转相除法), 见
 Greatest Common Divisor (最大公因子)
 evaluation (求值), 38-48, 136
 lazy (惰性), 见 call-by-need (传需调用)
 strict (严格), 见 call-by-value (传值调用)
 exception declarations (声明), 135-136, 304, 460
 exception specifications (描述), 310, 459
 exceptions (异常), 134-141, 462
 and commands (及 命令), 320-321
 eliminating (消除), 151
 type checking of (之 类型检测), 325
 exists function (函数), 184-185, 187
 exn type (类型), 135-139, 141, 177, 321
 exp function (函数), 24

explode function (函数), 73
 expressions (表达式), 462-463
 in programming languages (程序设计语言中的), 2-4

F

fact function (函数), 40-42, 245
facti function (函数), 40-42, 47
 proofs about (相关的证明), 214, 220-222, 245, 247
 type checking of (之类型检测), 64
 factorials (阶乘), 40-43, 189, 317-319, 又见 *fact*, *facti*
 in λ -calculus (λ -演算中的), 388-389, 393-395
Fail exception (异常), 138
false constructor (构造子), 26, 127
 Fibonacci numbers (斐波那契数), 49-51, 191, 222-223, 329-330
filter function (函数), 182-183, 187, 209
 for sequences (序列上的), 196, 206
 Fitzgerald, J. S., 256
 fixed point property (不动点性质), 389, 392
FixedInt structure (结构), 14
floor function (函数), 24
fn expressions (表达式), 172-174, 178, 323, 427-428, 462
 and delayed evaluation (及延时求值), 193, 202, 391
foldl function (函数), 185-187
 proofs about (相关的证明), 236-237
foldr function (函数), 185-187, 190, 211, 409
 proofs about (相关的证明), 237
 formulæ (公式), 398
 in ML (用ML书写的), 408
 Fortran (Fortran语言), 2, 7, 9, 127, 356
 FP (FP语言), 9
from function (函数), 193, 199
 Frost, R., 371
 fully-functorial programming (全函子式程序设计), 294-299
fun declarations (声明), 19-20, 28-31, 125-127, 460
 of curried functions (柯里函数的), 174, 182-183
 polymorphic (多态的), 325
 functional languages (函数式语言), 9
 functional programming (函数式程序设计), 1-11, 38, 58
 and imperative feature (及命令式特性), 327-330, 336-339
 application of (之应用), 10
 functionals (算子), 7-8, 179-190, 409, 426-428, 又见
 tacticals (策略算子), tactics (策略)
 and parsing (及语法分析), 362-366
 proofs about (相关的证明), 233-237
 functions (函数), 6
 as arguments (作为参数), 177-178, 280
 as data (作为数据), 176-177, 191-192
 curried (柯里的), 173-178, 183-185, 209
 declaring (声明), 见 *fun* declarations (声明)
 higher-order (高阶), 见 functionals (算子)

 iterative (迭代的), 42-44, 49, 51, 76-78, 151, 186, 247
 recursive (递归的), 6, 40-44, 48-53, 175, 317
 with multiple arguments/results (具有多个参数/结果),
 29-32, 82, 110
functor declarations (声明), 272, 275-277, 285-289, 312, 458
 functors (函子), 271-299, 309, 又见 fully-functorial
 programming (全函子式程序设计)

G

Gansner, Emden, 13
 garbage collection (垃圾收集), 5-6, 130, 313
 Gaussian elimination (高斯消元法), 90-92
General structure (结构), 15
 Gerhart, Susan, 256
 Gordon, Michael J. C., 12, 440, 443
 Grant, P. W., 92
 graphs (图), 102-107, 278-280
GREATER constructor (构造子), 127
 Greatest Common Divisor (最大公因子), 3, 10, 48, 53, 248
 for polynomials (多项式上的), 120-121
 Greiner, John, 326

H

Hal, 397, 407-443
 Halfant, Matthew, 200
 Hamming problem (海明问题), 330
handle keyword (关键字), 138, 462
 HARP system (HARP系统), 443
 Harper, Robert, 326
 Haskell (Haskell语言), 9, 10, 92, 102, 131注
hd function (函数), 74, 82, 138, 176
 for sequences (序列上的), 192, 327
 heaps (堆), 见 priority queues (优先队列)
 binomial (二项式), 164
 Hoare, C. A. R. (霍尔), 10, 15, 69, 110, 111
 HOL system (HOL系统), 443
 Holmström, Sören, 336
 Hoogerwoord, R., 159
 HOPE (HOPE语言), 12
 HTML (HTML语言), 348-350
 Hudak, Paul, 9
 Huet, Gérard, 12, 421
 Hughes, John, 200, 336

I

identifiers (标识符), 21-22, 61, 465-467
if expressions (表达式), 见 conditional expressions (条件表达式)
ignore function (函数), 319
 imperative programming (命令式程序设计), 2-5, 79, 108

in ML (ML中的), 313-340, 344-356
ImperativeIO functor (函子), 350
implode function (函数), 73
include specifications (描述), 307-308, 310, 459
 induction (归纳)
 on natural numbers (对于自然数的), 216-224, 244-245
 on size (对于大小的), 238-245
 structural (结构), 224-233, 245
 well-founded (良基), 238, 242-247
infix declarations (声明), 460
infix operators (中缀操作符), 36-38, 283, 303, 363, 462
 parsing (语法分析), 364-366, 412-414
 input/output (输入/输出), 8, 340-356
 instances (实例)
 of polymorphic types (多态类型的), 65, 176
 of signatures (签名的), 264
 of terms/formulae (项/公式的), 379, 416-420
Int structure (结构), 24, 340
int type (类型), 22-24
inter function (函数), 98, 183
interleave function (函数), 195, 202
IntInf structure (结构), 14
io exception (异常), 344
 Isabelle system (Isabelle系统), 246, 421, 440, 443
it value (值), 19, 50, 174
iterates function (函数), 196, 198, 331

K

keywords, of Standard ML (Standard ML的关键字), 21

L

Lakatos, Imre, 256
 λ -calculus (λ -演算), 182, 372-396
 LAMBDA system (LAMBDA系统), 13
 Landin, Peter, 12
 Launchbury, J., 371
 Lazy ML (惰性ML语言), 9
 LCF system (LCF系统), 11-13, 421, 440, 443
 leanTAP system (leanTAP系统), 443
 left-recursive rules (左递归规则), 361, 381, 412
length function (函数), 76-77, 82, 229
 Leroy, Xavier, 136
 LESS constructor (构造子), 127
let expressions (表达式), 53-55, 135-137, 300-301, 318
 and polymorphism (及多态性), 324
 Letz, R., 443
 lexical analysis (词法分析), 358-360, 368, 412
 library (库), xiii, 13-15, 127, 319
 arithmetic and (算术), 24, 303
 arrays and (数组), 335

characters and (字符), 26
 functionals and (算子), 187
 input/output and (输入/输出), 350
 lists and (表), 82, 335
 strings and (字符串), 26
 Lisp (Lisp语言), 7, 9, 75, 234
List structure (结构), 82, 138
list type (类型), 70, 144
ListPair structure (结构), 82, 187
 lists (表, 列表), 6, 69-122, 141, 336
 doubly linked (双向链接), 见 ring buffers (环形缓冲区)
 functionals for (之上的算子), 182-188, 195
 in λ -calculus (λ -演算中的), 387-388
 in other languages (其他语言中的), 71
 induction on (对于它的归纳), 225
 lazy (惰性), 见 sequences (序列)
ln function (函数), 24
 local declarations (声明), 55-56, 300-301, 457, 460
 logic (逻辑)
 first-order (一阶), 398-407
 notation (符号), 215
 propositional (命题), 164-170, 399-400
 LOLITA (LOLITA系统), 11

M

Ménissier-Morain, Valérie, 446
 MacQueen, David, 12, 299
 Magnusson, Lena, 13
 making change (找零钱), 83-84, 139, 197-198
map function (函数), 182-183, 187-188, 190, 209
 for pairs of lists (表序偶上的), 187
 for sequences (序列上的), 195, 199
 proofs about (相关的证明), 235-236
Match exception (异常), 73, 137
Math structure (结构), 24, 137
 matrix operations (矩阵运算), 87-93, 183, 275-280
 Mauny, Michel, 136
max function (函数), 24
maxl function (函数), 72-73, 238
mem function (函数), 97-99, 185
 merging (合并), 111, 117-118
 meta-variables (元变量), 见 variables, in unification (合一中的变量)
 Miller, Keith, 108
 Miller Steven, 256
 Milner, Robin, 11-12, 66, 421
min function (函数), 24
 Miranda (Miranda语言), 9
 ML (ML语言), 1
 and verification (及验证), 214-215

as meta-language (作为元语言), 12-13, 430-431
 compilers for (它的编译器), xii-xiii
 evolution of (之演变), xiii, 11-12, 28
 Standard, 10-16

ML-Yacc (ML书写的编译器构造程序), 371

mod infix (中缀), 22, 49

modules (模块), 12, 16, 59-63, 257-312, 又见 functors (函数), signatures (签名), structures (结构)
 reference guide to (参考指南), 308-312

multisets (多重集合), 251-254, 399

names (名字), 见 identifiers (N标识符)

natural numbers (自然数), 216-219, 224

in λ -calculus (λ -演算中的), 386-387, 393-395

negation normal form (否定范式), 166-167, 238-240

newmem function (函数), 98, 187

Newton-Raphson method (牛顿-拉夫森方法), 见 square roots, real (实数平方根)

nil constructor (构造子), 70-71

Nipkow, Tobias, 371

nlength function (函数), 76, 226-227, 229, 233, 238

NONE constructor (构造子), 128

nonfix declarations (声明), 38, 460

Nordström, Bengt, 13

normal forms (范式)

in λ -calculus (λ -演算中的), 374-375, 392

not function (函数), 27, 127

NQTHM system (NQTHM系统), 246

nrev function (函数), 79-80, 227-233, 237

nth function (函数), 138, 424

null function (函数), 74, 82

for sequences (序列上的), 327

Nuprl system (Nuprl系统), 443

O

o infix (中缀), 180-181, 187, 234-237, 370

O'Keefe, Richard, 112-113

occurs check (出现检测), 416-417

Odersky, M., 102

Okasaki, Chris, 159, 164

op keyword (关键字), 37-38, 176-177, 180-181, 186, 465

open declarations (声明), 299-305, 363, 460

Oppacher, F., 443

Oppen, Derek, 354

option type (类型), 128

ord function (函数), 25-26

ORDER signature (签名), 280-284, 286

order type (类型), 127, 281

ordered binary decision diagrams (有序二叉决策图), 170, 446

ordered predicate (谓词), 249

orelse keyword (关键字), 27, 43, 462

OS structure (结构), 15

Otter system (Otter系统), 443

Overflow exception (异常), 137

overloading (重载), 23-24, 27, 64, 67, 102

P

pairs and tuples (序偶和元组), 27-38, 136

in λ -calculus (λ -演算中的), 386

palindromes (回文), 207-208, 211

parameters, in logic (逻辑中的参数), 406-407, 416-417, 428-433, 435

Park, Stephen, 108

parsing (语法分析), 360-372, 381-382, 411-414
 LR, 371

Pascal (Pascal语言), 4, 7, 108, 175

pointers in (之中的指针), 316, 321, 332

patterns (模式), 130-133, 182-183, 463-464

and fn notation (及 fn 记法), 172

for datatypes (数据类型上的), 125-127

for lists (表上的), 72-73

for pairs and tuples (序偶和元组上的), 28-32

for records (记录上的), 34-35, 338

for trees (树上的), 143

in val declarations (val声明中的), 31-32, 131-132, 138

layered (分层的), 见 as keyword (关键字)

non-exhaustive (未穷尽的), 73-75, 83-85, 137

overlapping (重叠), 126, 168

wildcard (通配符), 74, 126, 131

Paulson, Lawrence C., 246, 440, 443

PDP-8 (PDP-8计算机), 446

Pelletier, F., 441

permutations (排列), 95-96, 251

Peyton Jones, Simon L., 10

polymorphism (多态性), 见 types (类型)

polysomials (多项式), 114-121, 446

Posegga, J., 443

powers (幂), 48-49, 223-224

powerset (幂集), 99-100

pretty printing (美化打印), 351-356, 369-371, 382-384, 414

Pretty structure (结构), 355

prime numbers (素数, 质数), 94, 199, 219

PrimIO functor (函子), 351

print function (函数), 345, 348

priority queues (优先队列), 159-164, 281, 283-284, 290-293

PriorityQueue functor (函子), 284, 296

product types (积类型), 27-38

products, Cartesian (笛卡尔积, 叉积), 100, 187, 203

Prolog (Prolog语言), 71, 417, 443

proof (证明), 399-407

automated (自动的), 440-443

constructive (构造性的), 219
 states (状态), 420-424
prop type (类型), 165

Q

Quaife, A., 443
 quantifiers (量词), 215-216, 403-407, 428-430
 and induction (及归纳), 221-224, 227-228, 245
 queues (队列), 206, 258-274
 mutable (可变更), 334
 priority (优先级), 见 *priority queues* (优先队列)

R

raise keyword (关键字), 136, 462
 random numbers (随机数), 108-109, 198-199
 Reade, Chris, 154
real function (函数), 24
REAL signature (签名), 303
Real structure (结构), 24, 340
real type (类型), 22-24, 303
Real32 structure (结构), 14
Real64 structure (结构), 14
rec keyword (关键字), 460
 records (记录), 32-35, 338
 selecting fields of (之域的选择), 34-35, 464, 467
 recursion (递归), 见 *datatype* declarations (声明);
 functions (函数)
 in λ -calculus (λ -演算中的), 388-395
 infinite (无穷), 363
 mutual (相互), 57
 well-founded (良基), 245-246
ref constructor (构造子), 314, 325
ref type (类型), 314, 321
 references (引用), 314-334
 and polymorphism (及多态性), 321-326
 cyclic (循环), 316, 329, 331
 referential transparency (引用透明), 3-4
reflect function (函数), 144, 189, 230-233
repeat function (函数)
 for parsing (语法分析上的), 362-363, 382
 for powers of a function (对于函数的幂), 188-189, 202
 tactical (策略算子), 436-437, 439-440
 Reppy, John, 13
rev function (函数), 79-80, 82
revAppend function (函数), 80, 227-228
 reversing a list (翻转表), 79-80, 186, 324-325, 又见 *nrev*,
 rev, *revAppend*
 ring buffers (环形缓冲区), 331-334

S

scanning (扫描), 见 *lexical analysis* (词法分析)

Scheme (Scheme语言), 9
 Scott, Dana, 13, 375注
 search (搜索), 204-211
 best-first (最佳优先), 160, 206
 breadth-first (广度优先), 104, 204-208, 210-211
 depth-first (深度优先), 103-105, 204-210, 437-443, 又
 见 *making change* (找零钱)
 iterative deepening (迭代深化), 210-211, 439, 441
 sections (片断), 179-181
 Sedgewick, Robert, 105, 108
 semirings (半环), 280
 sequences (序列), 191-211, 366-367, 423
 and numerical analysis (及数值分析), 199-201
 infinite (无穷), 8, 195, 226
 in λ -calculus (λ -演算中的), 388, 389, 395
 using references (使用引用), 327-330
 sequents (相继式), 398-407, 428
 basic (基本的), 399, 423-424
 set operations (集合运算), 98-100, 187
 SETHEO system (SETHEO系统), 443
 sharing (共享)
 constraints (约束), 290-294, 306-310, 459
 of references (引用的), 323
 side effects (副作用), 3, 323
sign function (函数), 24
 signature constraints (签名约束), 264-266, 269, 275, 299, 311-312
 signature declarations (声明), 62-63, 311, 457
 signatures (签名), 62-63, 263-271, 285, 309-310, 458
 closed (闭合的), 297-299
 empty (空), 288
sin function (函数), 24
 Sisal (Sisal语言), 9
 Size exception (异常), 138
size function (函数), 25
 for trees (树上的), 143, 189, 232
 Skolem functions (Skolem函数), 417
 Sleator, Daniel, 262
 Smith, M. H., 11
 software development (软件开发), 15-16, 213, 285
SOME constructor (构造子), 128
 sorting (排序), 108-114, 160-162, 177-178
 topological (拓扑), 105-107
 verification of (之验证), 249-254
 space (空间), 5, 42, 130
 specifications (描述)
 executable (可执行的), 10, 229
 in signatures (签名中的), 264, 286, 309-310, 459, 又见
 sharing constrains (共享约束)
 of programs (程序的), 220, 248-249, 251, 255-256
sqrt function (函数), 24

square roots (平方根)
 integer (整数), 52-53
 real (实数), 54-55, 199-201, 又见 *sqrt* function (函数)
 Srivas, Mandayam, 256
 static binding (静态绑定), 21
 Stickel, Mark, 170, 446
str function (函数), 26
StreamIO functor (函子), 350
 streams, input/output (输入输出流), 344-347
String structure (结构), 26, 341-342
string type (类型), 26
 • *StringCvt* structure (结构), 341
 strings (字符串), 24-26, 73-74, 184, 340-343, 465-466
 structure declarations (声明), 60-63, 311, 457
 structure specifications (描述), 283-284, 310, 459
 structures (结构), 60-62, 308, 311, 458
 empty (空), 288
 in logic (逻辑中的), 398
subs infix (中缀), 99, 115
Subscript exception (异常), 82, 137
 substitution (替换)
 in λ -calculus (λ -演算中的), 373-379
 in logic (逻辑中的), 403-404, 408
Substring structure (结构), 26, 341, 348
 substrings (子字符串, 子串), 26, 341-342, 348-350
 Suen, E., 443
 sum of two squares (两个平方之和), 93-94
 Sussman, Gerald, 200

T

tacticals (策略算子), 13, 436-440
 tactics (策略), 13, 420-435, 440-443
take function (函数), 77, 82, 111, 424
 for sequences (序列上的), 193, 328
 proofs about (相关的证明), 251-254
takewhile function (函数), 184
 Tarditi, David, 371
 Tarjan, Robert, 262
 tautology checking (重言式检测), 164-170, 238-242, 354
 terms (项)
 in logic (逻辑中的), 398, 408
 of λ -calculus (λ -演算中的), 372-373, 378-379
 theorem proving (定理证明), 11-13, 397-443, 446
Time structure (结构), 15
Timer structure (结构), 15
tl function (函数), 75, 82, 138
 for sequences (序列上的), 192, 327
 Tofte, Mads, 12, 325
TREE signature (签名), 295, 297
Tree structure (结构), 148, 297, 304-305

tree type (类型), 142, 148, 304
 trees (树), 6, 141-164, 189
 balanced (平衡), 143-147, 150-154
 binary search (二叉搜索), 150-154, 176-177, 281-283
 induction on (之上的归纳), 229
 traversing (遍历), 145-147, 189, 231-233
true constructor (构造子), 26, 127
trunc function (函数), 24
 Turner, David A., 9, 47, 100, 182
 type abbreviations (类型缩写), 见 *type* declarations (声明)
 in signatures (签名中的), 307
 type constraints (类型约束), 23-24, 29-30, 35, 324, 334, 462
 type constructors (类型构造子), 368
 type declarations (声明), 29, 35, 460
 type specifications (描述), 269, 309, 459
 type variables (类型变量), 65-67, 368, 466
 equality (相等), 97-99
 types (类型), 63-67, 199, 464
 dynamic (动态), 136, 177
 equality (相等), 97, 102
 parsing and displaying (语法分析及显示), 367-371
 polymorphic (多态), 128-130, 321-326
 restrictiveness of (之限制性), 71, 124, 325

U

unification (合一), 405-407, 416-420
 efficient (高效的), 420
 higher-order (高阶), 421
 tactic for (有关的策略), 423-426, 435
union function (函数), 98, 115
unit type (类型), 32, 129-130, 192, 321
 universe (整体), 398
unzip function (函数), 81-82
 Uribe, T., 170
 user interfaces (用户界面), 431

V

v-arrays (v-数组, 版本树数组), 336-339
 val declarations (声明), 18-19, 31-32, 172-174, 428, 460
 polymorphic (多态), 178, 323-324
 val specifications (描述), 309, 459
 validity (有效性), 398-399
 variables (变量), 21
 in λ -calculus (λ -演算中的), 372-373, 375-377, 379
 in logic (逻辑中的), 398, 408
 in unification (合一中的), 405-407, 416-417, 428-430, 434-435
 type (类型), 见 *type* variables (类型变量)
Vector structure (结构), 335
 vectors (向量), 27-32, 37, 89
 verification (验证), 13, 213-256

limitations of (之 局限), 223, 254-256

W

Wegner, Peter, 67

well-founded relations (良基关系), 242-244, 246

where type qualification (限定), 310

while expressions (表达式), 318, 462

withtype keyword (关键字), 460

Word8 structure (结构), 14

Wright, Andrew, 326

Y

Y combinator (组合子), 389, 391-392

Z

Zhang, Hantao (张瀚涛), 170

zip function (函数), 81, 82

预定义标识符

`datatype bool = true | false`

布尔数据类型

`not : bool -> bool`

`= <> : 'a * 'a -> bool`

`type int and real`

整数和实数类型

`~ : num -> num`

`+ - * : num * num -> num`

`abs : num -> num`

`/ : real * real -> real`

`div mod : int * int -> int`

`<> <= >= : numtext * numtext -> bool`

`real : int -> real`

`round : real -> int`

`floor : real -> int`

`ceil : real -> int`

`trunc : real -> int`

`type char and string and substring`

字符、字符串和子串类型

`^ : string * string -> string`

`concat : string list -> string`

`explode : string -> char list`

`implode : char list -> string`

`str : char -> string`

`size : string -> int`

`substring : string * int * int -> string`

`chr : int -> char`

`ord : char -> int`

`datatype 'a list = nil | :: of 'a * 'a list`

表的数据类型

`@ : 'a list * 'a list -> 'a list`

`foldl foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b`

真值

逻辑非

相等测试

数

取负 (`num = int`或`real`)

加法、减法、乘法

绝对值

实数除法

整数除法的商和余数

关系 (`numtext = int`、`real`、`char`或`string`)

转换到最近似的实数

转换到最近似的整数

转换到(不大于源的)最大整数

转换到(不小于源的)最小整数

转换到绝对值(不大于源的符号相同的)最大整数

字符/字符串类型

连接两个字符串

连接一个表中的所有字符串

转换到字符的表

转换到字符串

转换到一个字符的字符串

字符串中的字符数

给定位置和大小子串

给定ASCII码的字符

字符的ASCII码

表

连接两个表

表的递归

<i>length</i>	: 'a list -> int	表的长度
<i>map</i>	: ('a -> 'b) -> 'a list -> 'b list	对表的成员应用一个函数
<i>rev</i>	: 'a list -> 'a list	表的翻转
<i>hd</i>	: 'a list -> 'a	表首
<i>tl</i>	: 'a list -> 'a list	表尾
<i>null</i>	: 'a list -> bool	空表的测试
<i>o</i>	: ('b->'c) * ('a->'b) -> 'a -> 'c	函数的复合
datatype 'a option = NONE SOME of 'a		可选值
可选数据类型		
<i>isSome</i>	: 'a option -> bool	测试SOME
<i>valOf</i>	: 'a option -> 'a	SOME的逆操作
<i>getOpt</i>	: 'a option * 'a -> 'a	从SOME中提取
datatype order = LESS EQUAL GREATER		序的关系
序的数据类型		
type unit		空元组的类型
单元类型		
<i>app</i>	: ('a -> unit) -> 'a list -> unit	对表的成员应用一个命令
<i>ignore</i>	: 'a -> list	转换到单元类型
<i>print</i>	: string -> unit	打印到终端
type 'a ref		引用
引用类型		
<i>!</i>	: 'a ref -> 'a	引用的内容
<i>ref</i>	: 'a -> 'a ref	引用的创建
<i>:=</i>	: 'a ref * 'a -> unit	引用的赋值
<i>before</i>	: 'a * 'b -> 'a	第一个操作数
type 'a array		可变更数组
数组类型		
type 'a vector		不可变数组
向量类型		
<i>vector</i>	: 'a list -> 'a vector	转换到向量
type exn		异常的类型
异常类型		
<i>exnName</i>	: exn -> string	异常的名字
<i>exnMessage</i>	: exn -> string	异常的信息
异常		
<i>Bind</i>		在val声明中无匹配

<i>Chr</i>	字符编码溢出
<i>Div</i>	除数为零
<i>Domain</i>	Math函数参数错误
<i>Fail of string</i>	一般错误
<i>Match</i>	在fun、case等中没有匹配的模式
<i>Option</i>	需要SOME
<i>Overflow</i>	算术运算溢出
<i>Size</i>	负数或过大的大小
<i>Subscript</i>	索引值(下标)溢出

中缀操作符的优先级 (除::和@之外都是左结合的)

7	/ * div mod
6	+ - ^
5	:: @
4	= <> < > <= >=
3	:= o
0	before